4. Memcheck: a memory error detector

Table of Contents

To use this tool, you may specify `--tool=memcheck` on the Valgrind command line. You don't have to, though, since Memcheck is the default tool.

4.1. Overview

Memcheck is a memory error detector. It can detect the following problems that are common in C and C++ programs.

- Accessing memory you shouldn't, e.g. overrunning and underrunning heap blocks, overrunning the top of the stack, and accessing memory after it has been freed.
- Using undefined values, i.e. values that have not been initialised, or that have been derived from other undefined values.
- Incorrect freeing of heap memory, such as double-freeing heap blocks, or mismatched use of `malloc`/`new`/`new[]` versus `free`/`delete`/`delete[]`

  Mismatches will also be reported for `sized` and `aligned` allocation and deallocation functions if the deallocation value does not match the allocation value.

- Overlapping `src` and `dst` pointers in `memcpy` and related functions.
- Passing a fishy (presumably negative) value to the `size` parameter of a memory allocation function.
- Using a `size` value of 0 with realloc.
- Using an `alignment` value that is not a power of two.
- Memory leaks.

Problems like these can be difficult to find by other means, often remaining undetected for long periods, then causing occasional, difficult-to-diagnose crashes.

Memcheck also provides Execution Trees memory profiling using the command line option `--xtree-memory` and the monitor command `xtmemory`.

4.2. Explanation of error messages from Memcheck

Memcheck issues a range of error messages. This section presents a quick summary of what error messages mean. The precise behaviour of the error-checking machinery is described in Details of Memcheck's checking machinery.

4.2.1. Illegal read / Illegal write errors

For example:

```
Invalid read of size 4
   at 0x40F6BBCC: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40F6B804: (within /usr/lib/libpng.so.2.1.0.9)
   by 0x40B07FF4: read_png_image(QImageIO *) (kernel/qpngio.cpp:326)
   by 0x40AC751B: QImageIO::read() (kernel/qimage.cpp:3621)
 Address 0xBFFFF0E0 is not stack'd, malloc'd or free'd
```

This happens when your program reads or writes memory at a place which Memcheck reckons it shouldn't. In this example, the program did a 4-byte read at address 0xBFFFF0E0, somewhere within the system-supplied library libpng.so.2.1.0.9, which was called from somewhere else in the same library, called from line 326 of `qpngio.cpp`, and so on.

Memcheck tries to establish what the illegal address might relate to, since that's often useful. So, if it points into a block of memory which has already been freed, you'll be informed of this, and also where the block was freed. Likewise, if it should turn out to be just off the end of a heap block, a common result of off-by-one-errors in array subscripting, you'll be informed of this fact, and also where the block was allocated. If you use the `--read-var-info` option Memcheck will run more slowly but may give a more detailed description of any illegal address.

In this example, Memcheck can't identify the address. Actually the address is on the stack, but, for some reason, this is not a valid stack address -- it is below the stack pointer and that isn't allowed. In this particular case it's probably caused by GCC generating invalid code, a known bug in some ancient versions of GCC.

Note that Memcheck only tells you that your program is about to access memory at an illegal address. It can't stop the access from happening. So, if your program makes an access which normally would result in a segmentation fault, you program will still suffer the same fate -- but you will get a message from Memcheck immediately prior to this. In this particular example, reading junk on the stack is non-fatal, and the program stays alive.

4.2.2. Use of uninitialised values

For example:

```
Conditional jump or move depends on uninitialised value(s)
   at 0x402DFA94: _IO_vfprintf (_itoa.h:49)
   by 0x402E8476: _IO_printf (printf.c:36)
   by 0x8048472: main (tests/manuel1.c:8)
```

An uninitialised-value use error is reported when your program uses a value which hasn't been initialised -- in other words, is undefined. Here, the undefined value is used somewhere inside the `printf` machinery of the C library. This error was reported when running the following small program:

```
int main()
{
  int x;
  printf ("x = %d\n", x);
}
```

It is important to understand that your program can copy around junk (uninitialised) data as much as it likes. Memcheck observes this and keeps track of the data, but does not complain. A complaint is issued only when your program attempts to make use of uninitialised data in a way that might affect your program's externally-visible behaviour. In this example, `x` is uninitialised. Memcheck observes the value being passed to `_IO_printf` and thence to `_IO_vfprintf`, but makes no comment. However, `_IO_vfprintf` has to examine the value of `x` so it can turn it into the corresponding ASCII string, and it is at this point that Memcheck complains.

Sources of uninitialised data tend to be:

- Local variables in procedures which have not been initialised, as in the example above.
- The contents of heap blocks (allocated with `malloc`, `new`, or a similar function) before you (or a constructor) write something there.

To see information on the sources of uninitialised data in your program, use the `--track-origins=yes` option. This makes Memcheck run more slowly, but can make it much easier to track down the root causes of uninitialised value errors.

4.2.3. Use of uninitialised or unaddressable values in system calls

Memcheck checks all parameters to system calls:

- It checks all the direct parameters themselves, whether they are initialised.
- Also, if a system call needs to read from a buffer provided by your program, Memcheck checks that the entire buffer is addressable and its contents are initialised.
- Also, if the system call needs to write to a user-supplied buffer, Memcheck checks that the buffer is addressable.

After the system call, Memcheck updates its tracked information to precisely reflect any changes in memory state caused by the system call.

Here's an example of two system calls with invalid parameters:

```
#include <stdlib.h>
#include <unistd.h>
int main( void )
{
  char* arr  = malloc(10);
  int*  arr2 = malloc(sizeof(int));
  write( 1 /* stdout */, arr, 10 );
  exit(arr2[0]);
}
```

You get these complaints ...

```
Syscall param write(buf) points to uninitialised byte(s)
   at 0x25A48723: __write_nocancel (in /lib/tls/libc-2.3.3.so)
   by 0x259AFAD3: __libc_start_main (in /lib/tls/libc-2.3.3.so)
   by 0x8048348: (within /auto/homes/njn25/grind/head4/a.out)
 Address 0x25AB8028 is 0 bytes inside a block of size 10 alloc'd
   at 0x259852B0: malloc (vg_replace_malloc.c:130)
   by 0x80483F1: main (a.c:5)

Syscall param exit(error_code) contains uninitialised byte(s)
   at 0x25A21B44: __GI__exit (in /lib/tls/libc-2.3.3.so)
   by 0x8048426: main (a.c:8)
```

... because the program has (a) written uninitialised junk from the heap block to the standard output, and (b) passed an uninitialised value to `exit`. Note that the first error refers to the memory pointed to by `buf` (not `buf` itself), but the second error refers directly to `exit`'s argument `arr2[0]`.

4.2.4. Illegal frees

For example:

```
Invalid free()
   at 0x4004FFDF: free (vg_clientmalloc.c:577)
   by 0x80484C7: main (tests/doublefree.c:10)
 Address 0x3807F7B4 is 0 bytes inside a block of size 177 free'd
```

```
   at 0x4004FFDF: free (vg_clientmalloc.c:577)
   by 0x80484C7: main (tests/doublefree.c:10)
```

Memcheck keeps track of the blocks allocated by your program with `malloc`/`new`, so it can know exactly whether or not the argument to `free`/`delete` is legitimate or not. Here, this test program has freed the same block twice. As with the illegal read/write errors, Memcheck attempts to make sense of the address freed. If, as here, the address is one which has previously been freed, you wil be told that -- making duplicate frees of the same block easy to spot. You will also get this message if you try to free a pointer that doesn't point to the start of a heap block.

4.2.5. When a heap block is freed with an inappropriate deallocation function

In the following example, a block allocated with `new[]` has wrongly been deallocated with `free`:

```
Mismatched free() / delete / delete []
   at 0x40043249: free (vg_clientfuncs.c:171)
   by 0x4102BB4E: QGArray::~QGArray(void) (tools/qgarray.cpp:149)
   by 0x4C261C41: PptDoc::~PptDoc(void) (include/qmemarray.h:60)
   by 0x4C261F0E: PptXml::~PptXml(void) (pptxml.cc:44)
 Address 0x4BB292A8 is 0 bytes inside a block of size 64 alloc'd
   at 0x4004318C: operator new[](unsigned int) (vg_clientfuncs.c:152)
   by 0x4C21BC15: KLaola::readSBStream(int) const (klaola.cc:314)
   by 0x4C21C155: KLaola::stream(KLaola::OLENode const *) (klaola.cc:416)
   by 0x4C21788F: OLEFilter::convert(QCString const &) (olefilter.cc:272)
```

In `C++` it's important to deallocate memory in a way compatible with how it was allocated.

Most of the time in C++ you will write code that uses `new expresions` and `delete expresions` (see cppreference new expression and cppreference delete expression). A new expression will call `operator new` to perform the allocation and then call the constructor (if one exists) on the object. Similarly a delete expression will call the destructor on the object (if one exists) and then call `operator delete`. The array overloads call constructors/destructors for each object in the array.

The deal is:

- If allocated with `malloc`, `calloc`, `realloc`, `valloc` or `memalign`, you must deallocate with `free`.
- If allocated with `new`, you must deallocate with `delete`.
- If allocated with `new[]`, you must deallocate with `delete[]`.

Mixing types of allocators and deallocators is undefined behaviour. That means that on some platforms you might not have any problems, but the same program may then crash on a different platform, Solaris for example. So it's best to fix it properly. According to the KDE folks "it's amazing how many C++ programmers don't know this".

The reason behind the requirement is as follows. In some C++ implementations, `delete[]` must be used for objects allocated by `new[]` because the compiler stores the size of the array and the pointer-to-member to the destructor of the array's content just before the pointer actually returned. `delete` doesn't account for this and will get confused, possibly corrupting the heap. Even if there is no corruption there are likely to be resource leaks since using the wrong delete may result in the wrong number of destructors being called.

C++ aligned allocations need to be freed using aligned delete with the same alignment.

4.2.6. Overlapping source and destination blocks

The following C library functions copy some data from one memory block to another (or something similar): `memcpy`, `strcpy`, `strncpy`, `strcat`, `strncat`. The blocks pointed to by their `src` and `dst` pointers aren't allowed to overlap. The POSIX standards have wording along the lines "If copying takes place between objects that overlap, the behavior is undefined." Therefore, Memcheck checks for this.

For example:

```
==27492== Source and destination overlap in memcpy(0xbffff294, 0xbffff280, 21)
==27492==    at 0x40026CDC: memcpy (mc_replace_strmem.c:71)
==27492==    by 0x804865A: main (overlap.c:40)
```

You don't want the two blocks to overlap because one of them could get partially overwritten by the copying.

You might think that Memcheck is being overly pedantic reporting this in the case where `dst` is less than `src`. For example, the obvious way to implement `memcpy` is by copying from the first byte to the last. However, the optimisation guides of some

architectures recommend copying from the last byte down to the first. Also, some implementations of `memcpy` zero `dst` before copying, because zeroing the destination's cache line(s) can improve performance.

The moral of the story is: if you want to write truly portable code, don't make any assumptions about the language implementation.

4.2.7. Fishy argument values

All memory allocation functions take an argument specifying the size of the memory block that should be allocated. Clearly, the requested size should be a non-negative value and is typically not excessively large. For instance, it is extremely unlikly that the size of an allocation request exceeds 2**63 bytes on a 64-bit machine. It is much more likely that such a value is the result of an erroneous size calculation and is in effect a negative value (that just happens to appear excessively large because the bit pattern is interpreted as an unsigned integer). Such a value is called a "fishy value". The `size` argument of the following allocation functions is checked for being fishy: `malloc`, `calloc`, `realloc`, `memalign`, `posix_memalign`, `aligned_alloc`, `new`, `new []`. `__builtin_new`, `__builtin_vec_new`, For `calloc` both arguments are checked.

For example:

```
==32233== Argument 'size' of function malloc has a fishy (possibly negative) value: -3
==32233==    at 0x4C2CFA7: malloc (vg_replace_malloc.c:298)
==32233==    by 0x400555: foo (fishy.c:15)
==32233==    by 0x400583: main (fishy.c:23)
```

In earlier Valgrind versions those values were being referred to as "silly arguments" and no back-trace was included.

4.2.8. Realloc size zero

The (ab)use or realloc to also do the job of `free` has been poorly understood for a long time. In the C17 standard ISO/IEC 9899:2017] the behaviour of realloc when the size argument is zero is specified as implementation defined. Memcheck warns about the non-portable use or realloc.

For example:

```
==77609== realloc() with size 0
==77609==    at 0x48502B8: realloc (vg_replace_malloc.c:1450)
==77609==    by 0x201989: main (realloczero.c:8)
==77609==  Address 0x5464040 is 0 bytes inside a block of size 4 alloc'd
==77609==    at 0x484CBB4: malloc (vg_replace_malloc.c:397)
==77609==    by 0x201978: main (realloczero.c:7)
```

4.2.9. Alignment Errors

C and C++ have several functions that allow the user to obtain aligned memory. Typically this is done for performance reasons so that the memory will be cache line or memory page aligned. C has the functions `memalign`, `posix_memalign` and `aligned_alloc`. C++ has numerous overloads of `operator new` and `operator delete`. Of these, posix_memalign is quite clearly specified, the others vary quite widely between implementations. Valgrind will generate errors for values of alignment that are invalid on any platform.

`memalign` will produce errors if the alignment is zero or not a multiple of two.

`posix_memalign` will produce errors if the alignment is less than sizeof(size_t), not a multiple of two or if the size is zero.

`aligned_alloc` will produce errors if the alignment is not a multiple of two , if the size is zero or if the size is not an integral multiple of the alignment.

`aligned new` will produce errors if the alignment is zero or not a multiple of two. The `nothrow` overloads will return a NULL pointer. The non-nothrow overloads will abort Valgrind.

`aligned delete` will produce errors if the alignment is zero or not a multiple of two or if the alignment is not the same as that used by `aligned new`.

`sized delete` will produce errors if the size is not the same as that used by `new`.

`sized aligned delete` combines the error conditions of the individual sized and aligned delete operators.

Example output:

```
==65825== Invalid alignment value: 3 (should be power of 2)
==65825==    at 0x485197E: memalign (vg_replace_malloc.c:1740)
```

```
==65825==    by 0x201CD2: main (memalign.c:39)
```

4.2.10. Memory leak detection

Memcheck keeps track of all heap blocks issued in response to calls to `malloc`/`new` et al. So when the program exits, it knows which blocks have not been freed.

If `--leak-check` is set appropriately, for each remaining block, Memcheck determines if the block is reachable from pointers within the root-set. The root-set consists of (a) general purpose registers of all threads, and (b) initialised, aligned, pointer-sized data words in accessible client memory, including stacks.

There are two ways a block can be reached. The first is with a "start-pointer", i.e. a pointer to the start of the block. The second is with an "interior-pointer", i.e. a pointer to the middle of the block. There are several ways we know of that an interior-pointer can occur:

- The pointer might have originally been a start-pointer and have been moved along deliberately (or not deliberately) by the program. In particular, this can happen if your program uses tagged pointers, i.e. if it uses the bottom one, two or three bits of a pointer, which are normally always zero due to alignment, in order to store extra information.
- It might be a random junk value in memory, entirely unrelated, just a coincidence.
- It might be a pointer to the inner char array of a C++ `std::string`. For example, some compilers add 3 words at the beginning of the std::string to store the length, the capacity and a reference count before the memory containing the array of characters. They return a pointer just after these 3 words, pointing at the char array.
- Some code might allocate a block of memory, and use the first 8 bytes to store (block size - 8) as a 64bit number. `sqlite3MemMalloc` does this.
- It might be a pointer to an array of C++ objects (which possess destructors) allocated with `new[]`. In this case, some compilers store a "magic cookie" containing the array length at the start of the allocated block, and return a pointer to just past that magic cookie, i.e. an interior-pointer. See this page for more information.
- It might be a pointer to an inner part of a C++ object using multiple inheritance.

You can optionally activate heuristics to use during the leak search to detect the interior pointers corresponding to the `stdstring`, `length64`, `newarray` and `multipleinheritance` cases. If the heuristic detects that an interior pointer corresponds to such a case, the block will be considered as reachable by the interior pointer. In other words, the interior pointer will be treated as if it were a start pointer.

With that in mind, consider the nine possible cases described by the following figure.

```
     Pointer chain            AAA Leak Case   BBB Leak Case
     -------------            -------------   -------------
(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

Leak Case 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
```

Every possible case can be reduced to one of the above nine. Memcheck merges some of these cases in its output, resulting in the following four leak kinds.

- "Still reachable". This covers cases 1 and 2 (for the BBB blocks) above. A start-pointer or chain of start-pointers to the block is found. Since the block is still pointed at, the programmer could, at least in principle, have freed it before program exit. "Still reachable" blocks are very common and arguably not a problem. So, by default, Memcheck won't report such blocks individually.

- "Definitely lost". This covers case 3 (for the BBB blocks) above. This means that no pointer to the block can be found. The block is classified as "lost", because the programmer could not possibly have freed it at program exit, since no pointer to it exists. This is likely a symptom of having lost the pointer at some earlier point in the program. Such cases should be fixed by the programmer.

- "Indirectly lost". This covers cases 4 and 9 (for the BBB blocks) above. This means that the block is lost, not because there are no pointers to it, but rather because all the blocks that point to it are themselves lost. For example, if you have a binary tree and the root node is lost, all its children nodes will be indirectly lost. Because the problem will disappear if the definitely lost block that caused the indirect leak is fixed, Memcheck won't report such blocks individually by default.

- "Possibly lost". This covers cases 5--8 (for the BBB blocks) above. This means that a chain of one or more pointers to the block has been found, but at least one of the pointers is an interior-pointer. This could just be a random value in memory that happens to point into a block, and so you shouldn't consider this ok unless you know you have interior-pointers.

(Note: This mapping of the nine possible cases onto four leak kinds is not necessarily the best way that leaks could be reported; in particular, interior-pointers are treated inconsistently. It is possible the categorisation may be improved in the future.)

Furthermore, if suppressions exists for a block, it will be reported as "suppressed" no matter what which of the above four kinds it belongs to.

The following is an example leak summary.

```
LEAK SUMMARY:
   definitely lost: 48 bytes in 3 blocks.
   indirectly lost: 32 bytes in 2 blocks.
     possibly lost: 96 bytes in 6 blocks.
   still reachable: 64 bytes in 4 blocks.
        suppressed: 0 bytes in 0 blocks.
```

If heuristics have been used to consider some blocks as reachable, the leak summary details the heuristically reachable subset of 'still reachable:' per heuristic. In the below example, of the 95 bytes still reachable, 87 bytes (56+7+8+16) have been considered heuristically reachable.

```
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: 95 bytes in 6 blocks
                       of which reachable via heuristic:
                         stdstring          : 56 bytes in 2 blocks
                         length64           : 16 bytes in 1 blocks
                         newarray           : 7 bytes in 1 blocks
                         multipleinheritance: 8 bytes in 1 blocks
        suppressed: 0 bytes in 0 blocks
```

If `--leak-check=full` is specified, Memcheck will give details for each definitely lost or possibly lost block, including where it was allocated. (Actually, it merges results for all blocks that have the same leak kind and sufficiently similar stack traces into a single "loss record". The `--leak-resolution` lets you control the meaning of "sufficiently similar".) It cannot tell you when or how or why the pointer to a leaked block was lost; you have to work that out for yourself. In general, you should attempt to ensure your programs do not have any definitely lost or possibly lost blocks at exit.

For example:

```
8 bytes in 1 blocks are definitely lost in loss record 1 of 14
   at 0x........: malloc (vg_replace_malloc.c:...)
   by 0x........: mk (leak-tree.c:11)
   by 0x........: main (leak-tree.c:39)

88 (8 direct, 80 indirect) bytes in 1 blocks are definitely lost in loss record 13 of 14
   at 0x........: malloc (vg_replace_malloc.c:...)
```

```
   by 0x........: mk (leak-tree.c:11)
   by 0x........: main (leak-tree.c:25)
```

The first message describes a simple case of a single 8 byte block that has been definitely lost. The second case mentions another 8 byte block that has been definitely lost; the difference is that a further 80 bytes in other blocks are indirectly lost because of this lost block. The loss records are not presented in any notable order, so the loss record numbers aren't particularly meaningful. The loss record numbers can be used in the Valgrind gdbserver to list the addresses of the leaked blocks and/or give more details about how a block is still reachable.

The option `--show-leak-kinds=<set>` controls the set of leak kinds to show when `--leak-check=full` is specified.

The `<set>` of leak kinds is specified in one of the following ways:

- a comma separated list of one or more of `definite indirect possible reachable`.
- `all` to specify the complete set (all leak kinds).
- `none` for the empty set.

The default value for the leak kinds to show is `--show-leak-kinds=definite,possible`.

To also show the reachable and indirectly lost blocks in addition to the definitely and possibly lost blocks, you can use `--show-leak-kinds=all`. To only show the reachable and indirectly lost blocks, use `--show-leak-kinds=indirect,reachable`. The reachable and indirectly lost blocks will then be presented as shown in the following two examples.

```
64 bytes in 4 blocks are still reachable in loss record 2 of 4
   at 0x........: malloc (vg_replace_malloc.c:177)
   by 0x........: mk (leak-cases.c:52)
   by 0x........: main (leak-cases.c:74)

32 bytes in 2 blocks are indirectly lost in loss record 1 of 4
   at 0x........: malloc (vg_replace_malloc.c:177)
   by 0x........: mk (leak-cases.c:52)
   by 0x........: main (leak-cases.c:80)
```

Because there are different kinds of leaks with different severities, an interesting question is: which leaks should be counted as true "errors" and which should not?

The answer to this question affects the numbers printed in the `ERROR SUMMARY` line, and also the effect of the `--error-exitcode` option. First, a leak is only counted as a true "error" if `--leak-check=full` is specified. Then, the option `--errors-for-leak-kinds=<set>` controls the set of leak kinds to consider as errors. The default value is `--errors-for-leak-kinds=definite,possible`

4.3. Memcheck Command-Line Options

`--leak-check=<no|summary|yes|full> [default: summary]`

> When enabled, search for memory leaks when the client program finishes. If set to `summary`, it says how many leaks occurred. If set to `full` or `yes`, each individual leak will be shown in detail and/or counted as an error, as specified by the options `--show-leak-kinds` and `--errors-for-leak-kinds`.
>
> If `--xml=yes` is given, memcheck will automatically use the value `--leak-check=full`. You can use `--show-leak-kinds=none` to reduce the size of the xml output if you are not interested in the leak results.

`--leak-resolution=<low|med|high> [default: high]`

> When doing leak checking, determines how willing Memcheck is to consider different backtraces to be the same for the purposes of merging multiple leaks into a single leak report. When set to `low`, only the first two entries need match. When `med`, four entries have to match. When `high`, all entries need to match.
>
> For hardcore leak debugging, you probably want to use `--leak-resolution=high` together with `--num-callers=40` or some such large number.
>
> Note that the `--leak-resolution` setting does not affect Memcheck's ability to find leaks. It only changes how the results are presented.

`--show-leak-kinds=<set> [default: definite,possible]`

> Specifies the leak kinds to show in a `full` leak search, in one of the following ways:
>
> - a comma separated list of one or more of `definite indirect possible reachable`.

- `all` to specify the complete set (all leak kinds). It is equivalent to `--show-leak-kinds=definite,indirect,possible,reachable`.
- `none` for the empty set.

`--errors-for-leak-kinds=<set> [default: definite,possible]`

Specifies the leak kinds to count as errors in a `full` leak search. The `<set>` is specified similarly to `--show-leak-kinds`

`--leak-check-heuristics=<set> [default: all]`

Specifies the set of leak check heuristics to be used during leak searches. The heuristics control which interior pointers to a block cause it to be considered as reachable. The heuristic set is specified in one of the following ways:

- a comma separated list of one or more of `stdstring length64 newarray multipleinheritance`.
- `all` to activate the complete set of heuristics. It is equivalent to `--leak-check-heuristics=stdstring,length64,newarray,multipleinheritance`.
- `none` for the empty set.

Note that these heuristics are dependent on the layout of the objects produced by the C++ compiler. They have been tested with some gcc versions (e.g. 4.4 and 4.7). They might not work properly with other C++ compilers.

`--show-reachable=<yes|no> , --show-possibly-lost=<yes|no>`

These options provide an alternative way to specify the leak kinds to show:

- `--show-reachable=no --show-possibly-lost=yes` is equivalent to `--show-leak-kinds=definite,possible`.
- `--show-reachable=no --show-possibly-lost=no` is equivalent to `--show-leak-kinds=definite`.
- `--show-reachable=yes` is equivalent to `--show-leak-kinds=all`.

Note that `--show-possibly-lost=no` has no effect if `--show-reachable=yes` is specified.

`--xtree-leak=<no|yes> [no]`

If set to yes, the results for the leak search done at exit will be output in a 'Callgrind Format' execution tree file. Note that this automatically sets the options `--leak-check=full` and `--show-leak-kinds=all`, to allow xtree visualisation tools such as kcachegrind to select what kind to leak to visualize. The produced file will contain the following events:

- `RB` : Reachable Bytes
- `PB` : Possibly lost Bytes
- `IB` : Indirectly lost Bytes
- `DB` : Definitely lost Bytes (direct plus indirect)
- `DIB` : Definitely Indirectly lost Bytes (subset of DB)
- `RBk` : reachable Blocks
- `PBk` : Possibly lost Blocks
- `IBk` : Indirectly lost Blocks
- `DBk` : Definitely lost Blocks

The increase or decrease for all events above will also be output in the file to provide the delta (increase or decrease) between 2 successive leak searches. For example, `iRB` is the increase of the `RB` event, `dPBk` is the decrease of `PBk` event. The values for the increase and decrease events will be zero for the first leak search done.

See Execution Trees for a detailed explanation about execution trees.

`--xtree-leak-file=<filename> [default: xtleak.kcg.%p]`

Specifies that Valgrind should produce the xtree leak report in the specified file. Any `%p`, `%q` or `%n` sequences appearing in the filename are expanded in exactly the same way as they are for `--log-file`. See the description of --log-file for details.

See Execution Trees for a detailed explanation about execution trees formats.

`--undef-value-errors=<yes|no> [default: yes]`

Controls whether Memcheck reports uses of undefined value errors. Set this to `no` if you don't want to see undefined value errors. It also has the side effect of speeding up Memcheck somewhat. AddrCheck (removed in Valgrind 3.1.0) functioned like Memcheck with `--undef-value-errors=no`.

`--track-origins=<yes|no> [default: no]`

Controls whether Memcheck tracks the origin of uninitialised values. By default, it does not, which means that although it can tell you that an uninitialised value is being used in a dangerous way, it cannot tell you where the uninitialised value came from. This often makes it difficult to track down the root problem.

When set to `yes`, Memcheck keeps track of the origins of all uninitialised values. Then, when an uninitialised value error is reported, Memcheck will try to show the origin of the value. An origin can be one of the following four places: a heap block, a stack allocation, a client request, or miscellaneous other sources (eg, a call to `brk`).

For uninitialised values originating from a heap block, Memcheck shows where the block was allocated. For uninitialised values originating from a stack allocation, Memcheck can tell you which function allocated the value, but no more than that -- typically it shows you the source location of the opening brace of the function. So you should carefully check that all of the function's local variables are initialised properly.

Performance overhead: origin tracking is expensive. It halves Memcheck's speed and increases memory use by a minimum of 100MB, and possibly more. Nevertheless it can drastically reduce the effort required to identify the root cause of uninitialised value errors, and so is often a programmer productivity win, despite running more slowly.

Accuracy: Memcheck tracks origins quite accurately. To avoid very large space and time overheads, some approximations are made. It is possible, although unlikely, that Memcheck will report an incorrect origin, or not be able to identify any origin.

Note that the combination `--track-origins=yes` and `--undef-value-errors=no` is nonsensical. Memcheck checks for and rejects this combination at startup.

`--partial-loads-ok=<yes|no> [default: yes]`

Controls how Memcheck handles 32-, 64-, 128- and 256-bit naturally aligned loads from addresses for which some bytes are addressable and others are not. When `yes`, such loads do not produce an address error. Instead, loaded bytes originating from illegal addresses are marked as uninitialised, and those corresponding to legal addresses are handled in the normal way.

When `no`, loads from partially invalid addresses are treated the same as loads from completely invalid addresses: an illegal-address error is issued, and the resulting bytes are marked as initialised.

Note that code that behaves in this way is in violation of the ISO C/C++ standards, and should be considered broken. If at all possible, such code should be fixed.

`--expensive-definedness-checks=<no|auto|yes> [default: auto]`

Controls whether Memcheck should employ more precise but also more expensive (time consuming) instrumentation when checking the definedness of certain values. In particular, this affects the instrumentation of integer adds, subtracts and equality comparisons.

Selecting `--expensive-definedness-checks=yes` causes Memcheck to use the most accurate analysis possible. This minimises false error rates but can cause up to 30% performance degradation.

Selecting `--expensive-definedness-checks=no` causes Memcheck to use the cheapest instrumentation possible. This maximises performance but will normally give an unusably high false error rate.

The default setting, `--expensive-definedness-checks=auto`, is strongly recommended. This causes Memcheck to use the minimum of expensive instrumentation needed to achieve the same false error rate as `--expensive-definedness-checks=yes`. It also enables an instrumentation-time analysis pass which aims to further reduce the costs of accurate instrumentation. Overall, the performance loss is generally around 5% relative to `--expensive-definedness-checks=no`, although this is strongly workload dependent. Note that the exact instrumentation settings in this mode are architecture dependent.

`--keep-stacktraces=alloc|free|alloc-and-free|alloc-then-free|none [default: alloc-and-free]`

Controls which stack trace(s) to keep for malloc'd and/or free'd blocks.

With `alloc-then-free`, a stack trace is recorded at allocation time, and is associated with the block. When the block is freed, a second stack trace is recorded, and this replaces the allocation stack trace. As a result, any "use after free" errors relating to this block can only show a stack trace for where the block was freed.

With `alloc-and-free`, both allocation and the deallocation stack traces for the block are stored. Hence a "use after free" error will show both, which may make the error easier to diagnose. Compared to `alloc-then-free`, this setting slightly increases Valgrind's memory use as the block contains two references instead of one.

With `alloc`, only the allocation stack trace is recorded (and reported). With `free`, only the deallocation stack trace is recorded (and reported). These values somewhat decrease Valgrind's memory and cpu usage. They can be useful depending on the error types you are searching for and the level of detail you need to analyse them. For example, if you are only interested in memory leak errors, it is sufficient to record the allocation stack traces.

With `none`, no stack traces are recorded for malloc and free operations. If your program allocates a lot of blocks and/or allocates/frees from many different stack traces, this can significantly decrease cpu and/or memory required. Of course, few details will be reported for errors related to heap blocks.

Note that once a stack trace is recorded, Valgrind keeps the stack trace in memory even if it is not referenced by any block. Some programs (for example, recursive algorithms) can generate a huge number of stack traces. If Valgrind uses too much memory in such circumstances, you can reduce the memory required with the options `--keep-stacktraces` and/or by using a smaller value for the option `--num-callers`.

If you want to use `--xtree-memory=full` memory profiling (see Execution Trees), then you cannot specify `--keep-stacktraces=free` or `--keep-stacktraces=none`.

`--freelist-vol=<number> [default: 20000000]`

When the client program releases memory using `free` (in `C`) or `delete` (`C++`), that memory is not immediately made available for re-allocation. Instead, it is marked inaccessible and placed in a queue of freed blocks. The purpose is to defer as long as possible the point at which freed-up memory comes back into circulation. This increases the chance that Memcheck will be able to detect invalid accesses to blocks for some significant period of time after they have been freed.

This option specifies the maximum total size, in bytes, of the blocks in the queue. The default value is twenty million bytes. Increasing this increases the total amount of memory used by Memcheck but may detect invalid uses of freed blocks which would otherwise go undetected.

`--freelist-big-blocks=<number> [default: 1000000]`

When making blocks from the queue of freed blocks available for re-allocation, Memcheck will in priority re-circulate the blocks with a size greater or equal to `--freelist-big-blocks`. This ensures that freeing big blocks (in particular freeing blocks bigger than `--freelist-vol`) does not immediately lead to a re-circulation of all (or a lot of) the small blocks in the free list. In other words, this option increases the likelihood to discover dangling pointers for the "small" blocks, even when big blocks are freed.

Setting a value of 0 means that all the blocks are re-circulated in a FIFO order.

`--workaround-gcc296-bugs=<yes|no> [default: no]`

When enabled, assume that reads and writes some small distance below the stack pointer are due to bugs in GCC 2.96, and does not report them. The "small distance" is 256 bytes by default. Note that GCC 2.96 is the default compiler on some ancient Linux distributions (RedHat 7.X) and so you may need to use this option. Do not use it if you do not have to, as it can cause real errors to be overlooked. A better alternative is to use a more recent GCC in which this bug is fixed.

You may also need to use this option when working with GCC 3.X or 4.X on 32-bit PowerPC Linux. This is because GCC generates code which occasionally accesses below the stack pointer, particularly for floating-point to/from integer conversions. This is in violation of the 32-bit PowerPC ELF specification, which makes no provision for locations below the stack pointer to be accessible.

This option is deprecated as of version 3.12 and may be removed from future versions. You should instead use `--ignore-range-below-sp` to specify the exact range of offsets below the stack pointer that should be ignored. A suitable equivalent is `--ignore-range-below-sp=1024-1`.

`--ignore-range-below-sp=<number>-<number>`

This is a more general replacement for the deprecated `--workaround-gcc296-bugs` option. When specified, it causes Memcheck not to report errors for accesses at the specified offsets below the stack pointer. The two offsets must be positive decimal numbers and -- somewhat counterintuitively -- the first one must be larger, in order to imply a non-wraparound address range to ignore. For example, to ignore 4 byte accesses at 8192 bytes below the stack pointer, use `--ignore-range-below-sp=8192-8189`. Only one range may be specified.

`--show-mismatched-frees=<yes|no> [default: yes]`

When enabled, Memcheck checks that heap blocks are deallocated using a function that matches the allocating function. That is, it expects `free` to be used to deallocate blocks allocated by `malloc`, `delete` for blocks allocated by `new`,

and `delete[]` for blocks allocated by `new[]`. If a mismatch is detected, an error is reported. This is in general important because in some environments, freeing with a non-matching function can cause crashes.

There is however a scenario where such mismatches cannot be avoided. That is when the user provides implementations of `new`/`new[]` that call `malloc` and of `delete`/`delete[]` that call `free`, and these functions are asymmetrically inlined. For example, imagine that `delete[]` is inlined but `new[]` is not. The result is that Memcheck "sees" all `delete[]` calls as direct calls to `free`, even when the program source contains no mismatched calls.

This causes a lot of confusing and irrelevant error reports. `--show-mismatched-frees=no` disables these checks. It is not generally advisable to disable them, though, because you may miss real errors as a result.

`--show-realloc-size-zero=<yes|no> [default: yes]`

When enabled, Memcheck checks for uses of `realloc` with a size of zero. This usage of `realloc` is unsafe since it is not portable. On some systems it will behave like `free`. On other systems it will either do nothing or else behave like a call to `free` followed by a call to `malloc` with a size of zero.

`--ignore-ranges=0xPP-0xQQ[,0xRR-0xSS]`

Any ranges listed in this option (and multiple ranges can be specified, separated by commas) will be ignored by Memcheck's addressability checking.

`--malloc-fill=<hexnumber>`

Fills blocks allocated by `malloc`, `new`, etc, but not by `calloc`, with the specified byte. This can be useful when trying to shake out obscure memory corruption problems. The allocated area is still regarded by Memcheck as undefined -- this option only affects its contents. Note that `--malloc-fill` does not affect a block of memory when it is used as argument to client requests VALGRIND_MEMPOOL_ALLOC or VALGRIND_MALLOCLIKE_BLOCK.

`--free-fill=<hexnumber>`

Fills blocks freed by `free`, `delete`, etc, with the specified byte value. This can be useful when trying to shake out obscure memory corruption problems. The freed area is still regarded by Memcheck as not valid for access -- this option only affects its contents. Note that `--free-fill` does not affect a block of memory when it is used as argument to client requests VALGRIND_MEMPOOL_FREE or VALGRIND_FREELIKE_BLOCK.

4.4. Writing suppression files

The basic suppression format is described in Suppressing errors.

The suppression-type (second) line should have the form:

```
Memcheck:suppression_type
```

The Memcheck suppression types are as follows:

- `Value1`, `Value2`, `Value4`, `Value8`, `Value16`, meaning an uninitialised-value error when using a value of 1, 2, 4, 8 or 16 bytes.
- `Cond` (or its old name, `Value0`), meaning use of an uninitialised CPU condition code.
- `Addr1`, `Addr2`, `Addr4`, `Addr8`, `Addr16`, meaning an invalid address during a memory access of 1, 2, 4, 8 or 16 bytes respectively.
- `Jump`, meaning an jump to an unaddressable location error.
- `Param`, meaning an invalid system call parameter error.
- `Free`, meaning an invalid or mismatching free.
- `Overlap`, meaning a `src` / `dst` overlap in `memcpy` or a similar function.
- `Leak`, meaning a memory leak.

`Param` errors have a mandatory extra information line at this point, which is the name of the offending system call parameter.

`Leak` errors have an optional extra information line, with the following format:

```
match-leak-kinds:<set>
```

where `<set>` specifies which leak kinds are matched by this suppression entry. `<set>` is specified in the same way as with the option `--show-leak-kinds`, that is, one of the following:

- a comma separated list of one or more of `definite indirect possible reachable`.
- `all` to specify the complete set (all leak kinds).
- `none` for the empty set.

If this optional extra line is not present, the suppression entry will match all leak kinds.

Be aware that leak suppressions that are created using `--gen-suppressions` will contain this optional extra line, and therefore may match fewer leaks than you expect. You may want to remove the line before using the generated suppressions.

The other Memcheck error kinds do not have extra lines.

If you give the `-v` option, Valgrind will print the list of used suppressions at the end of execution. For a leak suppression, this output gives the number of different loss records that match the suppression, and the number of bytes and blocks suppressed by the suppression. If the run contains multiple leak checks, the number of bytes and blocks are reset to zero before each new leak check. Note that the number of different loss records is not reset to zero.

In the example below, in the last leak search, 7 blocks and 96 bytes have been suppressed by a suppression with the name `some_leak_suppression`:

```
--21041-- used_suppression:     10 some_other_leak_suppression s.supp:14 suppressed: 12,400 bytes in 1 blocks
--21041-- used_suppression:     39 some_leak_suppression s.supp:2 suppressed: 96 bytes in 7 blocks
```

For `ValueN` and `AddrN` errors, the first line of the calling context is either the name of the function in which the error occurred, or, failing that, the full path of the `.so` file or executable containing the error location. For `Free` errors, the first line is the name of the function doing the freeing (eg, `free`, `__builtin_vec_delete`, etc). For `Overlap` errors, the first line is the name of the function with the overlapping arguments (eg. `memcpy`, `strcpy`, etc).

The last part of any suppression specifies the rest of the calling context that needs to be matched.

4.5. Details of Memcheck's checking machinery

Read this section if you want to know, in detail, exactly what and how Memcheck is checking.

4.5.1. Valid-value (V) bits

It is simplest to think of Memcheck implementing a synthetic CPU which is identical to a real CPU, except for one crucial detail. Every bit (literally) of data processed, stored and handled by the real CPU has, in the synthetic CPU, an associated "valid-value" bit, which says whether or not the accompanying bit has a legitimate value. In the discussions which follow, this bit is referred to as the V (valid-value) bit.

Each byte in the system therefore has a 8 V bits which follow it wherever it goes. For example, when the CPU loads a word-size item (4 bytes) from memory, it also loads the corresponding 32 V bits from a bitmap which stores the V bits for the process' entire address space. If the CPU should later write the whole or some part of that value to memory at a different address, the relevant V bits will be stored back in the V-bit bitmap.

In short, each bit in the system has (conceptually) an associated V bit, which follows it around everywhere, even inside the CPU. Yes, all the CPU's registers (integer, floating point, vector and condition registers) have their own V bit vectors. For this to work, Memcheck uses a great deal of compression to represent the V bits compactly.

Copying values around does not cause Memcheck to check for, or report on, errors. However, when a value is used in a way which might conceivably affect your program's externally-visible behaviour, the associated V bits are immediately checked. If any of these indicate that the value is undefined (even partially), an error is reported.

Here's an (admittedly nonsensical) example:

```
int i, j;
int a[10], b[10];
for ( i = 0; i < 10; i++ ) {
  j = a[i];
  b[i] = j;
}
```

Memcheck emits no complaints about this, since it merely copies uninitialised values from `a[]` into `b[]`, and doesn't use them in a way which could affect the behaviour of the program. However, if the loop is changed to:

```
for ( i = 0; i < 10; i++ ) {
  j += a[i];
```

```
}
if ( j == 77 )
  printf("hello there\n");
```

then Memcheck will complain, at the `if`, that the condition depends on uninitialised values. Note that it **doesn't** complain at the `j += a[i];`, since at that point the undefinedness is not "observable". It's only when a decision has to be made as to whether or not to do the `printf` -- an observable action of your program -- that Memcheck complains.

Most low level operations, such as adds, cause Memcheck to use the V bits for the operands to calculate the V bits for the result. Even if the result is partially or wholly undefined, it does not complain.

Checks on definedness only occur in three places: when a value is used to generate a memory address, when control flow decision needs to be made, and when a system call is detected, Memcheck checks definedness of parameters as required.

If a check should detect undefinedness, an error message is issued. The resulting value is subsequently regarded as well-defined. To do otherwise would give long chains of error messages. In other words, once Memcheck reports an undefined value error, it tries to avoid reporting further errors derived from that same undefined value.

This sounds overcomplicated. Why not just check all reads from memory, and complain if an undefined value is loaded into a CPU register? Well, that doesn't work well, because perfectly legitimate C programs routinely copy uninitialised values around in memory, and we don't want endless complaints about that. Here's the canonical example. Consider a struct like this:

```
struct S { int x; char c; };
struct S s1, s2;
s1.x = 42;
s1.c = 'z';
s2 = s1;
```

The question to ask is: how large is `struct S`, in bytes? An `int` is 4 bytes and a `char` one byte, so perhaps a `struct S` occupies 5 bytes? Wrong. All non-toy compilers we know of will round the size of `struct S` up to a whole number of words, in this case 8 bytes. Not doing this forces compilers to generate truly appalling code for accessing arrays of `struct S`'s on some architectures.

So `s1` occupies 8 bytes, yet only 5 of them will be initialised. For the assignment `s2 = s1`, GCC generates code to copy all 8 bytes wholesale into `s2` without regard for their meaning. If Memcheck simply checked values as they came out of memory, it would yelp every time a structure assignment like this happened. So the more complicated behaviour described above is necessary. This allows GCC to copy `s1` into `s2` any way it likes, and a warning will only be emitted if the uninitialised values are later used.

As explained above, Memcheck maintains 8 V bits for each byte in your process, including for bytes that are in shared memory. However, the same piece of shared memory can be mapped multiple times, by several processes or even by the same process (for example, if the process wants a read-only and a read-write mapping of the same page). For such multiple mappings, Memcheck tracks the V bits for each mapping independently. This can lead to false positive errors, as the shared memory can be initialised via a first mapping, and accessed via another mapping. The access via this other mapping will have its own V bits, which have not been changed when the memory was initialised via the first mapping. The bypass for these false positives is to use Memcheck's client requests `VALGRIND_MAKE_MEM_DEFINED` and `VALGRIND_MAKE_MEM_UNDEFINED` to inform Memcheck about what your program does (or what another process does) to these shared memory mappings.

4.5.2. Valid-address (A) bits

Notice that the previous subsection describes how the validity of values is established and maintained without having to say whether the program does or does not have the right to access any particular memory location. We now consider the latter question.

As described above, every bit in memory or in the CPU has an associated valid-value (V) bit. In addition, all bytes in memory, but not in the CPU, have an associated valid-address (A) bit. This indicates whether or not the program can legitimately read or write that location. It does not give any indication of the validity of the data at that location -- that's the job of the V bits -- only whether or not the location may be accessed.

Every time your program reads or writes memory, Memcheck checks the A bits associated with the address. If any of them indicate an invalid address, an error is emitted. Note that the reads and writes themselves do not change the A bits, only consult them.

So how do the A bits get set/cleared? Like this:

- When the program starts, all the global data areas are marked as accessible.
- When the program does `malloc`/`new`, the A bits for exactly the area allocated, and not a byte more, are marked as accessible. Upon freeing the area the A bits are changed to indicate inaccessibility.

- When the stack pointer register (`SP`) moves up or down, A bits are set. The rule is that the area from `SP` up to the base of the stack is marked as accessible, and below `SP` is inaccessible. (If that sounds illogical, bear in mind that the stack grows down, not up, on almost all Unix systems, including GNU/Linux.) Tracking `SP` like this has the useful side-effect that the section of stack used by a function for local variables etc is automatically marked accessible on function entry and inaccessible on exit.

- When doing system calls, A bits are changed appropriately. For example, `mmap` magically makes files appear in the process' address space, so the A bits must be updated if `mmap` succeeds.

- Optionally, your program can tell Memcheck about such changes explicitly, using the client request mechanism described above.

4.5.3. Putting it all together

Memcheck's checking machinery can be summarised as follows:

- Each byte in memory has 8 associated V (valid-value) bits, saying whether or not the byte has a defined value, and a single A (valid-address) bit, saying whether or not the program currently has the right to read/write that address. As mentioned above, heavy use of compression means the overhead is typically around 25%.

- When memory is read or written, the relevant A bits are consulted. If they indicate an invalid address, Memcheck emits an Invalid read or Invalid write error.

- When memory is read into the CPU's registers, the relevant V bits are fetched from memory and stored in the simulated CPU. They are not consulted.

- When a register is written out to memory, the V bits for that register are written back to memory too.

- When values in CPU registers are used to generate a memory address, or to determine the outcome of a conditional branch, the V bits for those values are checked, and an error emitted if any of them are undefined.

- When values in CPU registers are used for any other purpose, Memcheck computes the V bits for the result, but does not check them.

- Once the V bits for a value in the CPU have been checked, they are then set to indicate validity. This avoids long chains of errors.

- When values are loaded from memory, Memcheck checks the A bits for that location and issues an illegal-address warning if needed. In that case, the V bits loaded are forced to indicate Valid, despite the location being invalid.

  This apparently strange choice reduces the amount of confusing information presented to the user. It avoids the unpleasant phenomenon in which memory is read from a place which is both unaddressable and contains invalid values, and, as a result, you get not only an invalid-address (read/write) error, but also a potentially large set of uninitialised-value errors, one for every time the value is used.

  There is a hazy boundary case to do with multi-byte loads from addresses which are partially valid and partially invalid. See details of the option `--partial-loads-ok` for details.

Memcheck intercepts calls to `malloc`, `calloc`, `realloc`, `valloc`, `memalign`, `free`, `new`, `new[]`, `delete` and `delete[]`. The behaviour you get is:

- `malloc`/`new`/`new[]`: the returned memory is marked as addressable but not having valid values. This means you have to write to it before you can read it.

- `calloc`: returned memory is marked both addressable and valid, since `calloc` clears the area to zero.

- `realloc`: if the new size is larger than the old, the new section is addressable but invalid, as with `malloc`. If the new size is smaller, the dropped-off section is marked as unaddressable. You may only pass to `realloc` a pointer previously issued to you by `malloc`/`calloc`/`realloc`.

- `free`/`delete`/`delete[]`: you may only pass to these functions a pointer previously issued to you by the corresponding allocation function. Otherwise, Memcheck complains. If the pointer is indeed valid, Memcheck marks the entire area it points at as unaddressable, and places the block in the freed-blocks-queue. The aim is to defer as long as possible reallocation of this block. Until that happens, all attempts to access it will elicit an invalid-address error, as you would hope.

4.6. Memcheck Monitor Commands

The Memcheck tool provides monitor commands handled by Valgrind's built-in gdbserver (see Monitor command handling by the Valgrind gdbserver). Valgrind python code provides GDB front end commands giving an easier usage of the memcheck monitor commands (see GDB front end commands for Valgrind gdbserver monitor commands). To launch a memcheck

monitor command via its GDB front end command, instead of prefixing the command with "monitor", you must use the GDB `memcheck` command (or the shorter aliases `mc`). Using the memcheck GDB front end command provide a more flexible usage, such as evaluation of address and length arguments by GDB. In GDB, you can use `help memcheck` to get help about the memcheck front end monitor commands and you can use `apropos memcheck` to get all the commands mentionning the word "memcheck" in their name or on-line help.

- `xb <addr> [<len>]` shows the definedness (V) bits and values for <len> (default 1) bytes starting at <addr>. For each 8 bytes, two lines are output.

  The first line shows the validity bits for 8 bytes. The definedness of each byte in the range is given using two hexadecimal digits. These hexadecimal digits encode the validity of each bit of the corresponding byte, using 0 if the bit is defined and 1 if the bit is undefined. If a byte is not addressable, its validity bits are replaced by __ (a double underscore).

  The second line shows the values of the bytes below the corresponding validity bits. The format used to show the bytes data is similar to the GDB command 'x /<len>xb <addr>'. The value for a non addressable bytes is shown as ?? (two question marks).

  In the following example, `string10` is an array of 10 characters, in which the even numbered bytes are undefined. In the below example, the byte corresponding to `string10[5]` is not addressable.

```
(gdb) p &string10
$4 = (char (*)[10]) 0x804a2f0
(gdb) mo xb 0x804a2f0 10
                  ff      00      ff      00      ff      __      ff      00
0x804A2F0:      0x3f    0x6e    0x3f    0x65    0x3f    0x??    0x3f    0x65
                  ff      00
0x804A2F8:      0x3f    0x00
Address 0x804A2F0 len 10 has 1 bytes unaddressable
(gdb)
```

  The GDB memcheck front end command `memcheck xb ADDR [LEN]` accepts any address expression for its first ADDR argument. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space. The following example shows how to get the definedness of `string10` using the memcheck xb front end command.

```
(gdb) mc xb &string10 sizeof(string10)
                  ff      00      ff      00      ff      __      ff      00
0x804A2F0:      0x3f    0x6e    0x3f    0x65    0x3f    0x??    0x3f    0x65
                  ff      00
0x804A2F8:      0x3f    0x00
Address 0x804A2F0 len 10 has 1 bytes unaddressable
(gdb)
```

  The command xb cannot be used with registers. To get the validity bits of a register, you must start Valgrind with the option `--vgdb-shadow-registers=yes`. The validity bits of a register can then be obtained by printing the 'shadow 1' corresponding register. In the below x86 example, the register eax has all its bits undefined, while the register ebx is fully defined.

```
(gdb) p /x $eaxs1
$9 = 0xffffffff
(gdb) p /x $ebxs1
$10 = 0x0
(gdb)
```

- `get_vbits <addr> [<len>]` shows the definedness (V) bits for <len> (default 1) bytes starting at <addr> using the same convention as the `xb` command. `get_vbits` only shows the V bits (grouped by 4 bytes). It does not show the values. If you want to associate V bits with the corresponding byte values, the `xb` command will be easier to use, in particular on little endian computers when associating undefined parts of an integer with their V bits values.

  The following example shows the result of `get_vbits` on the `string10` used in the `xb` command explanation. The GDB memcheck equivalent front end command `memcheck get_vbits ADDR [LEN]`accepts any ADDR expression and any LEN expression (separated by a space).

```
(gdb) monitor get_vbits 0x804a2f0 10
ff00ff00 ff__ff00 ff00
Address 0x804A2F0 len 10 has 1 bytes unaddressable
(gdb) memcheck get_vbits &string10 sizeof(string10)
ff00ff00 ff__ff00 ff00
Address 0x804A2F0 len 10 has 1 bytes unaddressable
```

- `make_memory [noaccess|undefined|defined|Definedifaddressable] <addr> [<len>]` marks the range of <len> (default 1) bytes at <addr> as having the given status. Parameter `noaccess` marks the range as non-accessible, so Memcheck will report an error on any access to it. `undefined` or `defined` mark the area as accessible, but Memcheck regards the bytes in it respectively as having undefined or defined values. `Definedifaddressable` marks as defined, bytes in the range which are already addressible, but makes no change to the status of bytes in the range which are not addressible. Note that the first letter of `Definedifaddressable` is an uppercase D to avoid confusion with `defined`.

  The GDB equivalent memcheck front end commands `memcheck make_memory [noaccess|undefined|defined|Definedifaddressable] ADDR [LEN]` accept any address expression for their first ADDR argument. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space.

  In the following example, the first byte of the `string10` is marked as defined and then is marked noaccess:

```
(gdb) monitor make_memory defined 0x8049e28  1
(gdb) monitor get_vbits 0x8049e28 10
0000ff00 ff00ff00 ff00
(gdb) memcheck make_memory noaccess &string10[0]
(gdb) memcheck get_vbits &string10 sizeof(string10)
__00ff00 ff00ff00 ff00
Address 0x8049E28 len 10 has 1 bytes unaddressable
(gdb)
```

- `check_memory [addressable|defined] <addr> [<len>]` checks that the range of <len> (default 1) bytes at <addr> has the specified accessibility. It then outputs a description of <addr>. In the following example, a detailed description is available because the option `--read-var-info=yes` was given at Valgrind startup:

```
(gdb) monitor check_memory defined 0x8049e28  1
Address 0x8049E28 len 1 defined
==14698==  Location 0x8049e28 is 0 bytes inside string10[0],
==14698==  declared at prog.c:10, in frame #0 of thread 1
(gdb)
```

  The GDB equivalent memcheck front end commands `memcheck check_memory [addressable|defined] ADDR [LEN]` accept any address expression for their first ADDR argument. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space.

- `leak_check [full*|summary|xtleak] [kinds <set>|reachable|possibleleak*|definiteleak] [heuristics heur1,heur2,...] [new|increased*|changed|any] [unlimited*|limited <max_loss_records_output>]` performs a leak check. The `*` in the arguments indicates the default values.

  If the `[full*|summary|xtleak]` argument is `summary`, only a summary of the leak search is given; otherwise a full leak report is produced. A full leak report gives detailed information for each leak: the stack trace where the leaked blocks were allocated, the number of blocks leaked and their total size. When a full report is requested, the next two arguments further specify what kind of leaks to report. A leak's details are shown if they match both the second and third argument. A full leak report might output detailed information for many leaks. The nr of leaks for which information is output can be controlled using the `limited` argument followed by the maximum nr of leak records to output. If this maximum is reached, the leak search outputs the records with the biggest number of bytes.

  The value `xtleak` also produces a full leak report, but output it as an xtree in a file xtleak.kcg.%p.%n (see --log-file). See Execution Trees for a detailed explanation about execution trees formats. See --xtree-leak for the description of the events in a xtree leak file.

  The `kinds` argument controls what kind of blocks are shown for a `full` leak search. The set of leak kinds to show can be specified using a `<set>` similarly to the command line option `--show-leak-kinds`. Alternatively, the value `definiteleak` is equivalent to `kinds definite`, the value `possibleleak` is equivalent to `kinds definite,possible` : it will also show possibly leaked blocks, .i.e those for which only an interior pointer was found. The value `reachable` will show all block categories (i.e. is equivalent to `kinds all`).

  The `heuristics` argument controls the heuristics used during the leak search. The set of heuristics to use can be specified using a `<set>` similarly to the command line option `--leak-check-heuristics`. The default value for the `heuristics` argument is `heuristics none`.

  The `[new|increased*|changed|any]` argument controls what kinds of changes are shown for a `full` leak search. The value `increased` specifies that only block allocation stacks with an increased number of leaked bytes or blocks since the previous leak check should be shown. The value `changed` specifies that allocation stacks with any change since the previous leak check should be shown. The value `new` specifies to show only the block allocation stacks that are new since the previous leak search. The value `any` specifies that all leak entries should be shown, regardless of any increase or decrease. If `new` or

`increased` or `changed` are specified, the leak report entries will show the delta relative to the previous leak report and the new loss records will have a "new" marker (even when `increased` or `changed` were specified).

The following example shows usage of the `leak_check` monitor command on the `memcheck/tests/leak-cases.c` regression test. The first command outputs one entry having an increase in the leaked bytes. The second command is the same as the first command, but uses the abbreviated forms accepted by GDB and the Valgrind gdbserver. It only outputs the summary information, as there was no increase since the previous leak search.

```
(gdb) monitor leak_check full possibleleak increased
==19520== 16 (+16) bytes in 1 (+1) blocks are possibly lost in new loss record 9 of 12
==19520==    at 0x40070B4: malloc (vg_replace_malloc.c:263)
==19520==    by 0x80484D5: mk (leak-cases.c:52)
==19520==    by 0x804855F: f (leak-cases.c:81)
==19520==    by 0x80488E0: main (leak-cases.c:107)
==19520==
==19520== LEAK SUMMARY:
==19520==    definitely lost: 32 (+0) bytes in 2 (+0) blocks
==19520==    indirectly lost: 16 (+0) bytes in 1 (+0) blocks
==19520==      possibly lost: 32 (+16) bytes in 2 (+1) blocks
==19520==    still reachable: 96 (+16) bytes in 6 (+1) blocks
==19520==         suppressed: 0 (+0) bytes in 0 (+0) blocks
==19520== Reachable blocks (those to which a pointer was found) are not shown.
==19520== To see them, add 'reachable any' args to leak_check
==19520==
(gdb) mo l
==19520== LEAK SUMMARY:
==19520==    definitely lost: 32 (+0) bytes in 2 (+0) blocks
==19520==    indirectly lost: 16 (+0) bytes in 1 (+0) blocks
==19520==      possibly lost: 32 (+0) bytes in 2 (+0) blocks
==19520==    still reachable: 96 (+0) bytes in 6 (+0) blocks
==19520==         suppressed: 0 (+0) bytes in 0 (+0) blocks
==19520== Reachable blocks (those to which a pointer was found) are not shown.
==19520== To see them, add 'reachable any' args to leak_check
==19520==
(gdb)
```

Note that when using Valgrind's gdbserver, it is not necessary to rerun with `--leak-check=full --show-reachable=yes` to see the reachable blocks. You can obtain the same information without rerunning by using the GDB command `monitor leak_check full reachable any` (or, using abbreviation: `mo l f r a`).

The GDB equivalent memcheck front end command `memcheck leak_check` auto-completes the user input by providing the full list of keywords still relevant according to what is already typed. For example, if the "summary" keyword has been provided, the following TABs to auto-complete other items will not propose anymore "full" and "xtleak". Note that KIND and HEUR values are not part of auto-completed elements.

- `block_list <loss_record_nr>|<loss_record_nr_from>..<loss_record_nr_to> [unlimited*|limited <max_blocks>] [heuristics heur1,heur2,...]` shows the list of blocks belonging to `<loss_record_nr>` (or to the loss records range `<loss_record_nr_from>..<loss_record_nr_to>`). The nr of blocks to print can be controlled using the `limited` argument followed by the maximum nr of blocks to output. If one or more heuristics are given, only prints the loss records and blocks found via one of the given `heur1,heur2,...` heuristics.

  A leak search merges the allocated blocks in loss records : a loss record re-groups all blocks having the same state (for example, Definitely Lost) and the same allocation backtrace. Each loss record is identified in the leak search result by a loss record number. The `block_list` command shows the loss record information followed by the addresses and sizes of the blocks which have been merged in the loss record. If a block was found using an heuristic, the block size is followed by the heuristic.

  If a directly lost block causes some other blocks to be indirectly lost, the block_list command will also show these indirectly lost blocks. The indirectly lost blocks will be indented according to the level of indirection between the directly lost block and the indirectly lost block(s). Each indirectly lost block is followed by the reference of its loss record.

  The block_list command can be used on the results of a leak search as long as no block has been freed after this leak search: as soon as the program frees a block, a new leak search is needed before block_list can be used again.

  In the below example, the program leaks a tree structure by losing the pointer to the block A (top of the tree). So, the block A is directly lost, causing an indirect loss of blocks B to G. The first block_list command shows the loss record of A (a definitely lost block with address 0x4028028, size 16). The addresses and sizes of the indirectly lost blocks due to block A are shown below the block A. The second command shows the details of one of the indirect loss records output by the first command.

```
          A
        /   \
       B     C
      / \   / \
     D   E F   G
```

```
(gdb) bt
#0  main () at leak-tree.c:69
(gdb) monitor leak_check full any
==19552== 112 (16 direct, 96 indirect) bytes in 1 blocks are definitely lost in loss record 7 of 7
==19552==    at 0x40070B4: malloc (vg_replace_malloc.c:263)
==19552==    by 0x80484D5: mk (leak-tree.c:28)
==19552==    by 0x80484FC: f (leak-tree.c:41)
==19552==    by 0x8048856: main (leak-tree.c:63)
==19552==
==19552== LEAK SUMMARY:
==19552==    definitely lost: 16 bytes in 1 blocks
==19552==    indirectly lost: 96 bytes in 6 blocks
==19552==      possibly lost: 0 bytes in 0 blocks
==19552==    still reachable: 0 bytes in 0 blocks
==19552==         suppressed: 0 bytes in 0 blocks
==19552==
(gdb) monitor block_list 7
==19552== 112 (16 direct, 96 indirect) bytes in 1 blocks are definitely lost in loss record 7 of 7
==19552==    at 0x40070B4: malloc (vg_replace_malloc.c:263)
==19552==    by 0x80484D5: mk (leak-tree.c:28)
==19552==    by 0x80484FC: f (leak-tree.c:41)
==19552==    by 0x8048856: main (leak-tree.c:63)
==19552== 0x4028028[16]
==19552==   0x4028068[16] indirect loss record 1
==19552==      0x40280E8[16] indirect loss record 3
==19552==      0x4028128[16] indirect loss record 4
==19552==   0x40280A8[16] indirect loss record 2
==19552==      0x4028168[16] indirect loss record 5
==19552==      0x40281A8[16] indirect loss record 6
(gdb) mo b 2
==19552== 16 bytes in 1 blocks are indirectly lost in loss record 2 of 7
==19552==    at 0x40070B4: malloc (vg_replace_malloc.c:263)
==19552==    by 0x80484D5: mk (leak-tree.c:28)
==19552==    by 0x8048519: f (leak-tree.c:43)
==19552==    by 0x8048856: main (leak-tree.c:63)
==19552== 0x40280A8[16]
==19552==   0x4028168[16] indirect loss record 5
==19552==   0x40281A8[16] indirect loss record 6
(gdb)
```

- `who_points_at <addr> [<len>]` shows all the locations where a pointer to addr is found. If len is equal to 1, the command only shows the locations pointing exactly at addr (i.e. the "start pointers" to addr). If len is > 1, "interior pointers" pointing at the len first bytes will also be shown.

  The locations searched for are the same as the locations used in the leak search. So, `who_points_at` can a.o. be used to show why the leak search still can reach a block, or can search for dangling pointers to a freed block. Each location pointing at addr (or pointing inside addr if interior pointers are being searched for) will be described.

  The GDB equivalent memcheck front end command `memcheck who_points_at ADDR [LEN]` accept any address expression for its first ADDR argument. The second optional argument is any integer expression. Note that these 2 arguments must be separated by a space.

  In the below example, the pointers to the 'tree block A' (see example in command `block_list`) is shown before the tree was leaked. The descriptions are detailed as the option `--read-var-info=yes` was given at Valgrind startup. The second call shows the pointers (start and interior pointers) to block G. The block G (0x40281A8) is reachable via block C (0x40280a8) and register ECX of tid 1 (tid is the Valgrind thread id). It is "interior reachable" via the register EBX.

```
(gdb) monitor who_points_at 0x4028028
==20852== Searching for pointers to 0x4028028
==20852== *0x8049e20 points at 0x4028028
==20852==  Location 0x8049e20 is 0 bytes inside global var "t"
```

```
==20852==  declared at leak-tree.c:35
(gdb) monitor who_points_at 0x40281A8 16
==20852== Searching for pointers pointing in 16 bytes from 0x40281a8
==20852== *0x40280ac points at 0x40281a8
==20852==  Address 0x40280ac is 4 bytes inside a block of size 16 alloc'd
==20852==    at 0x40070B4: malloc (vg_replace_malloc.c:263)
==20852==    by 0x80484D5: mk (leak-tree.c:28)
==20852==    by 0x8048519: f (leak-tree.c:43)
==20852==    by 0x8048856: main (leak-tree.c:63)
==20852== tid 1 register ECX points at 0x40281a8
==20852== tid 1 register EBX interior points at 2 bytes inside 0x40281a8
(gdb)
```

When `who_points_at` finds an interior pointer, it will report the heuristic(s) with which this interior pointer will be considered as reachable. Note that this is done independently of the value of the option `--leak-check-heuristics`. In the below example, the loss record 6 indicates a possibly lost block. `who_points_at` reports that there is an interior pointer pointing in this block, and that the block can be considered reachable using the heuristic `multipleinheritance`.

```
(gdb) monitor block_list 6
==3748== 8 bytes in 1 blocks are possibly lost in loss record 6 of 7
==3748==    at 0x4007D77: operator new(unsigned int) (vg_replace_malloc.c:313)
==3748==    by 0x8048954: main (leak_cpp_interior.cpp:43)
==3748== 0x402A0E0[8]
(gdb) monitor who_points_at 0x402A0E0 8
==3748== Searching for pointers pointing in 8 bytes from 0x402a0e0
==3748== *0xbe8ee078 interior points at 4 bytes inside 0x402a0e0
==3748==  Address 0xbe8ee078 is on thread 1's stack
==3748== block at 0x402a0e0 considered reachable by ptr 0x402a0e4 using multipleinheritance heuristic
(gdb)
```

- `xtmemory [<filename> default xtmemory.kcg.%p.%n]` requests Memcheck tool to produce an xtree heap memory report. See Execution Trees for a detailed explanation about execution trees.

4.7. Client Requests

The following client requests are defined in `memcheck.h`. See `memcheck.h` for exact details of their arguments.

- `VALGRIND_MAKE_MEM_NOACCESS`, `VALGRIND_MAKE_MEM_UNDEFINED` and `VALGRIND_MAKE_MEM_DEFINED`. These mark address ranges as completely inaccessible, accessible but containing undefined data, and accessible and containing defined data, respectively. They return -1, when run on Valgrind and 0 otherwise.

- `VALGRIND_MAKE_MEM_DEFINED_IF_ADDRESSABLE`. This is just like `VALGRIND_MAKE_MEM_DEFINED` but only affects those bytes that are already addressable.

- `VALGRIND_CHECK_MEM_IS_ADDRESSABLE` and `VALGRIND_CHECK_MEM_IS_DEFINED`: check immediately whether or not the given address range has the relevant property, and if not, print an error message. Also, for the convenience of the client, returns zero if the relevant property holds; otherwise, the returned value is the address of the first byte for which the property is not true. Always returns 0 when not run on Valgrind.

- `VALGRIND_CHECK_VALUE_IS_DEFINED`: a quick and easy way to find out whether Valgrind thinks a particular value (lvalue, to be precise) is addressable and defined. Prints an error message if not. It has no return value.

- `VALGRIND_DO_LEAK_CHECK`: does a full memory leak check (like `--leak-check=full`) right now. This is useful for incrementally checking for leaks between arbitrary places in the program's execution. It has no return value.

- `VALGRIND_DO_ADDED_LEAK_CHECK`: same as `VALGRIND_DO_LEAK_CHECK` but only shows the entries for which there was an increase in leaked bytes or leaked number of blocks since the previous leak search. It has no return value.

- `VALGRIND_DO_CHANGED_LEAK_CHECK`: same as `VALGRIND_DO_LEAK_CHECK` but only shows the entries for which there was an increase or decrease in leaked bytes or leaked number of blocks since the previous leak search. It has no return value.

- `VALGRIND_DO_NEW_LEAK_CHECK`: same as `VALGRIND_DO_LEAK_CHECK` but only shows the new entries since the previous leak search. It has no return value.

- `VALGRIND_DO_QUICK_LEAK_CHECK`: like `VALGRIND_DO_LEAK_CHECK`, except it produces only a leak summary (like `--leak-check=summary`). It has no return value.

- `VALGRIND_COUNT_LEAKS`: fills in the four arguments with the number of bytes of memory found by the previous leak check to be leaked (i.e. the sum of direct leaks and indirect leaks), dubious, reachable and suppressed. This is useful in test harness code, after calling `VALGRIND_DO_LEAK_CHECK` or `VALGRIND_DO_QUICK_LEAK_CHECK`.

- `VALGRIND_COUNT_LEAK_BLOCKS`: identical to `VALGRIND_COUNT_LEAKS` except that it returns the number of blocks rather than the number of bytes in each category.

- `VALGRIND_GET_VBITS` and `VALGRIND_SET_VBITS`: allow you to get and set the V (validity) bits for an address range. You should probably only set V bits that you have got with `VALGRIND_GET_VBITS`. Only for those who really know what they are doing.

- `VALGRIND_CREATE_BLOCK` and `VALGRIND_DISCARD`. `VALGRIND_CREATE_BLOCK` takes an address, a number of bytes and a character string. The specified address range is then associated with that string. When Memcheck reports an invalid access to an address in the range, it will describe it in terms of this block rather than in terms of any other block it knows about. Note that the use of this macro does not actually change the state of memory in any way -- it merely gives a name for the range.

  At some point you may want Memcheck to stop reporting errors in terms of the block named by `VALGRIND_CREATE_BLOCK`. To make this possible, `VALGRIND_CREATE_BLOCK` returns a "block handle", which is a C `int` value. You can pass this block handle to `VALGRIND_DISCARD`. After doing so, Valgrind will no longer relate addressing errors in the specified range to the block. Passing invalid handles to `VALGRIND_DISCARD` is harmless.

4.8. Memory Pools: describing and working with custom allocators

Some programs use custom memory allocators, often for performance reasons. Left to itself, Memcheck is unable to understand the behaviour of custom allocation schemes as well as it understands the standard allocators, and so may miss errors and leaks in your program. What this section describes is a way to give Memcheck enough of a description of your custom allocator that it can make at least some sense of what is happening.

There are many different sorts of custom allocator, so Memcheck attempts to reason about them using a loose, abstract model. We use the following terminology when describing custom allocation systems:

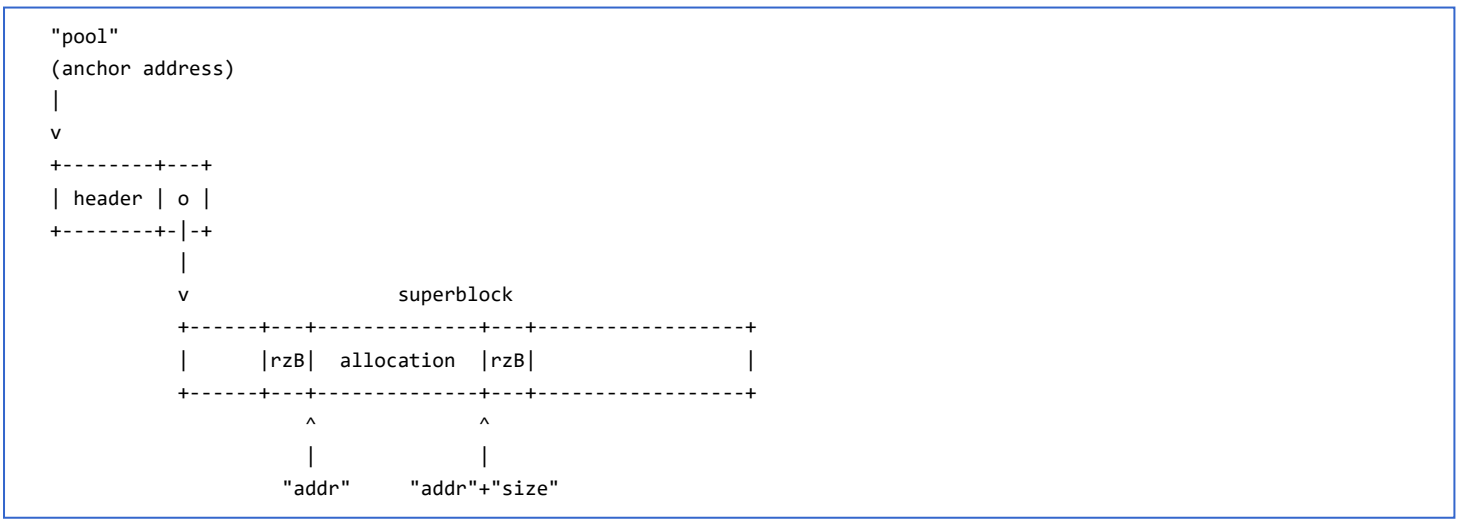
- Custom allocation involves a set of independent "memory pools".

- Memcheck's notion of a a memory pool consists of a single "anchor address" and a set of non-overlapping "chunks" associated with the anchor address.

- Typically a pool's anchor address is the address of a book-keeping "header" structure.

- Typically the pool's chunks are drawn from a contiguous "superblock" acquired through the system `malloc` or `mmap`.

Keep in mind that the last two points above say "typically": the Valgrind mempool client request API is intentionally vague about the exact structure of a mempool. There is no specific mention made of headers or superblocks. Nevertheless, the following picture may help elucidate the intention of the terms in the API:



Note that the header and the superblock may be contiguous or discontiguous, and there may be multiple superblocks associated with a single header; such variations are opaque to Memcheck. The API only requires that your allocation scheme can present sensible values of "pool", "addr" and "size".

Typically, before making client requests related to mempools, a client program will have allocated such a header and superblock for their mempool, and marked the superblock NOACCESS using the `VALGRIND_MAKE_MEM_NOACCESS` client request.

When dealing with mempools, the goal is to maintain a particular invariant condition: that Memcheck believes the unallocated portions of the pool's superblock (including redzones) are NOACCESS. To maintain this invariant, the client program must

ensure that the superblock starts out in that state; Memcheck cannot make it so, since Memcheck never explicitly learns about the superblock of a pool, only the allocated chunks within the pool.

Once the header and superblock for a pool are established and properly marked, there are a number of client requests programs can use to inform Memcheck about changes to the state of a mempool:

- `VALGRIND_CREATE_MEMPOOL(pool, rzB, is_zeroed)`: This request registers the address `pool` as the anchor address for a memory pool. It also provides a size `rzB`, specifying how large the redzones placed around chunks allocated from the pool should be. Finally, it provides an `is_zeroed` argument that specifies whether the pool's chunks are zeroed (more precisely: defined) when allocated.

  Upon completion of this request, no chunks are associated with the pool. The request simply tells Memcheck that the pool exists, so that subsequent calls can refer to it as a pool.

- `VALGRIND_CREATE_MEMPOOL_EXT(pool, rzB, is_zeroed, flags)`: Create a memory pool with some flags (that can be OR-ed together) specifying extended behaviour. When flags is zero, the behaviour is identical to `VALGRIND_CREATE_MEMPOOL`.

  - The flag `VALGRIND_MEMPOOL_METAPOOL` specifies that the pieces of memory associated with the pool using `VALGRIND_MEMPOOL_ALLOC` will be used by the application as superblocks to dole out MALLOC_LIKE blocks using `VALGRIND_MALLOCLIKE_BLOCK`. In other words, a meta pool is a "2 levels" pool : first level is the blocks described by `VALGRIND_MEMPOOL_ALLOC`. The second level blocks are described using `VALGRIND_MALLOCLIKE_BLOCK`. Note that the association between the pool and the second level blocks is implicit : second level blocks will be located inside first level blocks. It is necessary to use the `VALGRIND_MEMPOOL_METAPOOL` flag for such 2 levels pools, as otherwise valgrind will detect overlapping memory blocks, and will abort execution (e.g. during leak search).

  - `VALGRIND_MEMPOOL_AUTO_FREE`. Such a meta pool can also be marked as an 'auto free' pool using the flag `VALGRIND_MEMPOOL_AUTO_FREE`, which must be OR-ed together with the `VALGRIND_MEMPOOL_METAPOOL`. For an 'auto free' pool, `VALGRIND_MEMPOOL_FREE` will automatically free the second level blocks that are contained inside the first level block freed with `VALGRIND_MEMPOOL_FREE`. In other words, calling `VALGRIND_MEMPOOL_FREE` will cause implicit calls to `VALGRIND_FREELIKE_BLOCK` for all the second level blocks included in the first level block. Note: it is an error to use the `VALGRIND_MEMPOOL_AUTO_FREE` flag without the `VALGRIND_MEMPOOL_METAPOOL` flag.

- `VALGRIND_DESTROY_MEMPOOL(pool)`: This request tells Memcheck that a pool is being torn down. Memcheck then removes all records of chunks associated with the pool, as well as its record of the pool's existence. While destroying its records of a mempool, Memcheck resets the redzones of any live chunks in the pool to NOACCESS.

- `VALGRIND_MEMPOOL_ALLOC(pool, addr, size)`: This request informs Memcheck that a `size`-byte chunk has been allocated at `addr`, and associates the chunk with the specified `pool`. If the pool was created with nonzero `rzB` redzones, Memcheck will mark the `rzB` bytes before and after the chunk as NOACCESS. If the pool was created with the `is_zeroed` argument set, Memcheck will mark the chunk as DEFINED, otherwise Memcheck will mark the chunk as UNDEFINED.

- `VALGRIND_MEMPOOL_FREE(pool, addr)`: This request informs Memcheck that the chunk at `addr` should no longer be considered allocated. Memcheck will mark the chunk associated with `addr` as NOACCESS, and delete its record of the chunk's existence.

- `VALGRIND_MEMPOOL_TRIM(pool, addr, size)`: This request trims the chunks associated with `pool`. The request only operates on chunks associated with `pool`. Trimming is formally defined as:

  - All chunks entirely inside the range `addr..(addr+size-1)` are preserved.

  - All chunks entirely outside the range `addr..(addr+size-1)` are discarded, as though `VALGRIND_MEMPOOL_FREE` was called on them.

  - All other chunks must intersect with the range `addr..(addr+size-1)`; areas outside the intersection are marked as NOACCESS, as though they had been independently freed with `VALGRIND_MEMPOOL_FREE`.

  This is a somewhat rare request, but can be useful in implementing the type of mass-free operations common in custom LIFO allocators.

- `VALGRIND_MOVE_MEMPOOL(poolA, poolB)`: This request informs Memcheck that the pool previously anchored at address `poolA` has moved to anchor address `poolB`. This is a rare request, typically only needed if you `realloc` the header of a mempool.

  No memory-status bits are altered by this request.

- `VALGRIND_MEMPOOL_CHANGE(pool, addrA, addrB, size)`: This request informs Memcheck that the chunk previously allocated at address `addrA` within `pool` has been moved and/or resized, and should be changed to cover the region `addrB..(addrB+size-1)`. This is a rare request, typically only needed if you `realloc` a superblock or wish to extend a chunk without changing its memory-status bits.

No memory-status bits are altered by this request.

- `VALGRIND_MEMPOOL_EXISTS(pool)`: This request informs the caller whether or not Memcheck is currently tracking a mempool at anchor address `pool`. It evaluates to 1 when there is a mempool associated with that address, 0 otherwise. This is a rare request, only useful in circumstances when client code might have lost track of the set of active mempools.

4.9. Debugging MPI Parallel Programs with Valgrind

Memcheck supports debugging of distributed-memory applications which use the MPI message passing standard. This support consists of a library of wrapper functions for the `PMPI_*` interface. When incorporated into the application's address space, either by direct linking or by `LD_PRELOAD`, the wrappers intercept calls to `PMPI_Send`, `PMPI_Recv`, etc. They then use client requests to inform Memcheck of memory state changes caused by the function being wrapped. This reduces the number of false positives that Memcheck otherwise typically reports for MPI applications.

The wrappers also take the opportunity to carefully check size and definedness of buffers passed as arguments to MPI functions, hence detecting errors such as passing undefined data to `PMPI_Send`, or receiving data into a buffer which is too small.

Unlike most of the rest of Valgrind, the wrapper library is subject to a BSD-style license, so you can link it into any code base you like. See the top of `mpi/libmpiwrap.c` for license details.

4.9.1. Building and installing the wrappers

The wrapper library will be built automatically if possible. Valgrind's configure script will look for a suitable `mpicc` to build it with. This must be the same `mpicc` you use to build the MPI application you want to debug. By default, Valgrind tries `mpicc`, but you can specify a different one by using the configure-time option `--with-mpicc`. Currently the wrappers are only buildable with `mpicc`s which are based on GNU GCC or Intel's C++ Compiler.

Check that the configure script prints a line like this:

```
checking for usable MPI2-compliant mpicc and mpi.h... yes, mpicc
```

If it says `... no`, your `mpicc` has failed to compile and link a test MPI2 program.

If the configure test succeeds, continue in the usual way with `make` and `make install`. The final install tree should then contain `libmpiwrap-<platform>.so`.

Compile up a test MPI program (eg, MPI hello-world) and try this:

```
LD_PRELOAD=$prefix/lib/valgrind/libmpiwrap-<platform>.so   \
           mpirun [args] $prefix/bin/valgrind ./hello
```

You should see something similar to the following

```
valgrind MPI wrappers 31901: Active for pid 31901
valgrind MPI wrappers 31901: Try MPIWRAP_DEBUG=help for possible options
```

repeated for every process in the group. If you do not see these, there is an build/installation problem of some kind.

The MPI functions to be wrapped are assumed to be in an ELF shared object with soname matching `libmpi.so*`. This is known to be correct at least for Open MPI and Quadrics MPI, and can easily be changed if required.

4.9.2. Getting started

Compile your MPI application as usual, taking care to link it using the same `mpicc` that your Valgrind build was configured with.

Use the following basic scheme to run your application on Valgrind with the wrappers engaged:

```
MPIWRAP_DEBUG=[wrapper-args]                                \
   LD_PRELOAD=$prefix/lib/valgrind/libmpiwrap-<platform>.so   \
   mpirun [mpirun-args]                                     \
   $prefix/bin/valgrind [valgrind-args]                     \
   [application] [app-args]
```

As an alternative to `LD_PRELOAD`ing `libmpiwrap-<platform>.so`, you can simply link it to your application if desired. This should not disturb native behaviour of your application in any way.

4.9.3. Controlling the wrapper library

Environment variable `MPIWRAP_DEBUG` is consulted at startup. The default behaviour is to print a starting banner

```
valgrind MPI wrappers 16386: Active for pid 16386
valgrind MPI wrappers 16386: Try MPIWRAP_DEBUG=help for possible options
```

and then be relatively quiet.

You can give a list of comma-separated options in `MPIWRAP_DEBUG`. These are

- `verbose`: show entries/exits of all wrappers. Also show extra debugging info, such as the status of outstanding `MPI_Request`s resulting from uncompleted `MPI_Irecv`s.
- `quiet`: opposite of `verbose`, only print anything when the wrappers want to report a detected programming error, or in case of catastrophic failure of the wrappers.
- `warn`: by default, functions which lack proper wrappers are not commented on, just silently ignored. This causes a warning to be printed for each unwrapped function used, up to a maximum of three warnings per function.
- `strict`: print an error message and abort the program if a function lacking a wrapper is used.

If you want to use Valgrind's XML output facility (`--xml=yes`), you should pass `quiet` in `MPIWRAP_DEBUG` so as to get rid of any extraneous printing from the wrappers.

4.9.4. Functions

All MPI2 functions except `MPI_Wtick`, `MPI_Wtime` and `MPI_Pcontrol` have wrappers. The first two are not wrapped because they return a `double`, which Valgrind's function-wrap mechanism cannot handle (but it could easily be extended to do so). `MPI_Pcontrol` cannot be wrapped as it has variable arity: `int MPI_Pcontrol(const int level, ...)`

Most functions are wrapped with a default wrapper which does nothing except complain or abort if it is called, depending on settings in `MPIWRAP_DEBUG` listed above. The following functions have "real", do-something-useful wrappers:

```
PMPI_Send PMPI_Bsend PMPI_Ssend PMPI_Rsend

PMPI_Recv PMPI_Get_count

PMPI_Isend PMPI_Ibsend PMPI_Issend PMPI_Irsend

PMPI_Irecv
PMPI_Wait PMPI_Waitall
PMPI_Test PMPI_Testall

PMPI_Iprobe PMPI_Probe

PMPI_Cancel

PMPI_Sendrecv

PMPI_Type_commit PMPI_Type_free

PMPI_Pack PMPI_Unpack

PMPI_Bcast PMPI_Gather PMPI_Scatter PMPI_Alltoall
PMPI_Reduce PMPI_Allreduce PMPI_Op_create

PMPI_Comm_create PMPI_Comm_dup PMPI_Comm_free PMPI_Comm_rank PMPI_Comm_size

PMPI_Error_string
PMPI_Init PMPI_Initialized PMPI_Finalize
```

A few functions such as `PMPI_Address` are listed as `HAS_NO_WRAPPER`. They have no wrapper at all as there is nothing worth checking, and giving a no-op wrapper would reduce performance for no reason.

Note that the wrapper library itself can itself generate large numbers of calls to the MPI implementation, especially when walking complex types. The most common functions called are `PMPI_Extent`, `PMPI_Type_get_envelope`, `PMPI_Type_get_contents`, and `PMPI_Type_free`.

4.9.5. Types

MPI-1.1 structured types are supported, and walked exactly. The currently supported combiners are `MPI_COMBINER_NAMED`, `MPI_COMBINER_CONTIGUOUS`, `MPI_COMBINER_VECTOR`, `MPI_COMBINER_HVECTOR` `MPI_COMBINER_INDEXED`, `MPI_COMBINER_HINDEXED` and `MPI_COMBINER_STRUCT`. This should cover all MPI-1.1 types. The mechanism (function `walk_type`) should extend easily to cover MPI2 combiners.

MPI defines some named structured types (`MPI_FLOAT_INT`, `MPI_DOUBLE_INT`, `MPI_LONG_INT`, `MPI_2INT`, `MPI_SHORT_INT`, `MPI_LONG_DOUBLE_INT`) which are pairs of some basic type and a C `int`. Unfortunately the MPI specification makes it impossible to look inside these types and see where the fields are. Therefore these wrappers assume the types are laid out as `struct { float val; int loc; }` (for `MPI_FLOAT_INT`), etc, and act accordingly. This appears to be correct at least for Open MPI 1.0.2 and for Quadrics MPI.

If `strict` is an option specified in `MPIWRAP_DEBUG`, the application will abort if an unhandled type is encountered. Otherwise, the application will print a warning message and continue.

Some effort is made to mark/check memory ranges corresponding to arrays of values in a single pass. This is important for performance since asking Valgrind to mark/check any range, no matter how small, carries quite a large constant cost. This optimisation is applied to arrays of primitive types (`double`, `float`, `int`, `long`, `long long`, `short`, `char`, and `long double` on platforms where `sizeof(long double) == 8`). For arrays of all other types, the wrappers handle each element individually and so there can be a very large performance cost.

4.9.6. Writing new wrappers

For the most part the wrappers are straightforward. The only significant complexity arises with nonblocking receives.

The issue is that `MPI_Irecv` states the recv buffer and returns immediately, giving a handle (`MPI_Request`) for the transaction. Later the user will have to poll for completion with `MPI_Wait` etc, and when the transaction completes successfully, the wrappers have to paint the recv buffer. But the recv buffer details are not presented to `MPI_Wait` -- only the handle is. The library therefore maintains a shadow table which associates uncompleted `MPI_Request`s with the corresponding buffer address/count/type. When an operation completes, the table is searched for the associated address/count/type info, and memory is marked accordingly.

Access to the table is guarded by a (POSIX pthreads) lock, so as to make the library thread-safe.

The table is allocated with `malloc` and never `free`d, so it will show up in leak checks.

Writing new wrappers should be fairly easy. The source file is `mpi/libmpiwrap.c`. If possible, find an existing wrapper for a function of similar behaviour to the one you want to wrap, and use it as a starting point. The wrappers are organised in sections in the same order as the MPI 1.1 spec, to aid navigation. When adding a wrapper, remember to comment out the definition of the default wrapper in the long list of defaults at the bottom of the file (do not remove it, just comment it out).

4.9.7. What to expect when using the wrappers

The wrappers should reduce Memcheck's false-error rate on MPI applications. Because the wrapping is done at the MPI interface, there will still potentially be a large number of errors reported in the MPI implementation below the interface. The best you can do is try to suppress them.

You may also find that the input-side (buffer length/definedness) checks find errors in your MPI use, for example passing too short a buffer to `MPI_Recv`.

Functions which are not wrapped may increase the false error rate. A possible approach is to run with `MPI_DEBUG` containing `warn`. This will show you functions which lack proper wrappers but which are nevertheless used. You can then write wrappers for them.

A known source of potential false errors are the `PMPI_Reduce` family of functions, when using a custom (user-defined) reduction function. In a reduction operation, each node notionally sends data to a "central point" which uses the specified reduction function to merge the data items into a single item. Hence, in general, data is passed between nodes and fed to the reduction function, but the wrapper library cannot mark the transferred data as initialised before it is handed to the reduction function, because all that happens "inside" the `PMPI_Reduce` call. As a result you may see false positives reported in your reduction function.