Debugging Memory Leaks In C

Dynamic memory management in C

When an array is defined in C, its size remains fixed. If the array has *automatic storage duration* [because it is an *internal variable* defined at *function scope*], then it lives in the *stack*, coming alive when the function is called and dying when the function terminates. If the array has *static storage duration* [because it is an *external variable* defined at *file scope* or because it is an internal variable defined using `static` storage specifier], then it lives in the *data* or *bss* region of program memory, staying alive throughout the duration of program execution. We must decide at compile time how large the array will be, regardless of whether it lives in the stack or in the data area. If the program needs to hold a variable number of elements, such as traffic information [which is highly dynamic], we must declare an array large enough to hold the maximum number of vehicles the program would process. If we underestimate vehicle count, application source code must be recompiled with a larger count. If the count is overestimated, then memory will be wasted.

This document introduces a region of program memory, called the *heap*, specifically intended for producing data structures whose lifetimes are independent of the execution periods of the functions that create and manipulate them. *Dynamic memory allocation* is the process of obtaining segments of memory from the heap at run time for use by the program. In contrast to static and automatic storage durations, values stored in these memory segments are said to have *dynamic storage duration* - the lifetimes of these values are controlled by the program at run time. The ability to allocate and then deallocate memory allows an application to manage its memory more efficiently and with greater flexibility. Instead of having to allocate memory to accommodate the largest possible size for a data structure, only the actual amount required needs to be allocated. In this case, the Standard C library's *heap manager*, which provides library routines to build dynamic arrays, is used. This approach eliminates anticipating array bounds at compile time and makes programs allocate only as much memory as they need.

The Standard C library header file `<stdlib.h>` declares heap storage allocation functions that enable a programmer to obtain unnamed chunks of memory dynamically; that is while the program is executing. The prototypes of the storage allocation and deallocation functions are:

```
1  void * malloc(size_t size);
2  void * calloc(size_t count, size_t size);
3  void * realloc(void *ptr, size_t size);
4  void   free(void *ptr);
```

With functions `malloc` and `calloc`, storage can be allocated for data after determining how much storage is required. If the estimate proves to be high or low, the size of allocated storage can be changed with function `realloc`. Finally when done with the allocated storage, function `free` is called to release it for subsequent use by functions `malloc` or `calloc` or `realloc`.

All these functions operate with `void *` - that is, with pointers for which no type information is known. Recall that a `void *` is a general-purpose pointer used to hold references to any data type. Any pointer can be assigned to a `void *`. It can then be cast back to its original pointer type. When this happens the value will be equal to the original pointer value. Being able to specify such a "non-type" for these functions is probably the *raison d'etre* for inventing `void *` pointers. Because heap storage allocation and deallocation functions know nothing about the later use or

type of the to-be-stored object, they use `void *` to universally apply to all types and they use bytes to specify the storage size.

Memory allocation functions `malloc` and `calloc`

`malloc` allocates a region of heap memory large enough to hold an object whose size [as measured by the `sizeof` operator] is `size`.

`calloc` allocates a region of heap memory large enough to hold an array of `count` elements, each of size `size` [typically given by `sizeof` operator] and then clears the allocated memory to zero.

Both `malloc` and `calloc` return null pointers if the allocation request cannot be fulfilled because there simply may not be enough contiguous heap memory left. Therefore, as shown in the following code fragment, the returned pointer must always be checked before being used.

```
char str[] = "SeaToShiningC";
// dynamically allocate memory to store a copy of string str[]
char *pc = malloc(strlen(str) + 1);
if (pc == NULL) {
  printf("malloc of %lu bytes failed!!!\n", strlen(str)+1);
  exit(EXIT_FAILURE);
} else {
  strcpy(pc, str);
}
```

Both `malloc` and `calloc` guarantee that the returned pointer will be properly aligned to satisfy the alignment requirements of the data type to be stored at that memory address. For example, on most modern processors, `int`s must be stored at addresses which are a multiple of four while `double`s must be stored at addresses which are a multiple of eight.

Since arrays and pointers are displayed and processed identically internally, individual blocks of data can also be accessed using array syntax:

```
// allocate memory for 3 int objects with malloc returning the
// address of first int object
int* p = malloc(3*sizeof(int));
// use pointer offsets to access the 3 int objects using subscript operator
p[0] = 1; p[1] = 2; p[2] = 3;
printf("address of p: %p | second value: %d\n", p, *(p+1));
```

We can also use `malloc` or `calloc` in the same manner as before to dynamically allocate memory for one or more objects of structure type. After defining the struct `MyStruct` which contains a number of data primitives, function `calloc` creates a block of memory having four times the size of `MyStruct`. As can be seen, the various data elements can be accessed very conveniently.

```c
typedef struct {
  int i;
  double d;
  char a[5];
} MyStruct;

// allocate and zero-out memory for 4 objects of type MyStruct
MyStruct *p = calloc(4, sizeof(MyStruct));
// use pointer offsets to access individual objects and structure member
// operator to access individual data members of the structure object
p[0].i    = 1;
p[0].d    = 3.14159;
p[0].a[0] = 'a';
```

Memory allocation function `realloc`

Consider the following code fragment that allocates contiguous memory for 1024 `int` elements with each element initialized to zero.

```c
int const COUNT = 1024;

// allocate COUNT x sizeof(int) bytes of contiguous heap memory zeroed out
int *pi = calloc(COUNT, sizeof(int));
if (pi == NULL) {
  printf("malloc of %lu bytes failed!!!\n", COUNT*sizeof(int));
  exit(EXIT_FAILURE);
}

// do other stuff ...
```

Even with dynamic memory allocation, it is not always possible to predetermine the precise amount of storage to allocate. To handle this situation, `realloc` is available.

```c
void * realloc(void *ptr, size_t size);
```

This function takes a pointer to a memory region previously allocated by one of the storage allocations functions and changes its size while preserving its contents. The first argument to `realloc` is a pointer to the start of some previously allocated memory. This is important: the pointer must be a value that was returned by a previous call to `malloc`, `calloc`, or `realloc`, or it must be `NULL`. The use of any other argument will lead to undefined results. The second argument to `realloc` is the new total size of the allocated area, either smaller or larger than the original allocated space.

`realloc` returns a pointer to the start of the allocated space, which may be different from the original pointer to the allocated space provided as the first argument. If it does move the physical address, `realloc` will copy the data to the new place and return a pointer to it. This is an important consideration with variables that point to the allocated space and then `realloc()` is called to change its size. If `realloc` has to move the space, then pointers will now be pointing into deallocated space, space that may be reclaimed with subsequent calls to `malloc`, `calloc`, or `realloc`. It is the programmer's responsibility to check the pointer returned by `realloc` and to adjust pointer variables if the data area has been moved.

Consider the previous code fragment, which allocated memory for `COUNT` number of `int` elements. If it is later discovered that twice as many integers are required, the following call to `realloc` will do the trick.

```
int *pi = realloc(pi, sizeof(int) * COUNT * 2);
```

Always keep in mind that the new value returned by `realloc` may not be equal to the old value of `pi`.

And, always check the value returned by `realloc` against `NULL` to protect against the scenario of the allocation request being unsatisfied.

Special cases for `realloc`

There are two special cases of `realloc` that one must be aware of. If supplied a `NULL` first argument, `realloc` will work like `malloc`. So the call

```
int *pi = realloc(NULL, sizeof(int) * COUNT * 2);
```

is similar to this call:

```
int *pi = malloc(sizeof(int) * COUNT * 2);
```

The second special case is when the second argument to `realloc` is zero. In that case, `realloc` is being asked to shrink some previously allocated space down to zero; in other words to release it. Here, `realloc` works just like `free`.

Memory deallocation function `free`

`free` takes a single argument that points to the start of a previously allocated area. The entire storage area is deallocated and is reused by subsequent allocation calls. If a program is using a lot of allocated memory, then giving back storage when done with the allocated memory will prevent the program's runtime dynamic memory storage requirements from exceeding the size of the heap. So to remove the entire array of integers pointed to by `pi`, the call

```
free(pi);
```

can be used, as can the call

```
realloc(pi, 0);
```

Dynamic memory is error-prone!!!

Manipulating heap memory is prone to the same bugs as manipulating stack memory:

- Dereferencing uninitialized pointers
- Dereferencing `NULL` pointers
- Reading uninitialized memory
- Off-by-one array subscripting
- Getting hosed by a malformed character string that doesn't have a null terminating character

and some new ones:

- Failing to `free` allocated memory causing a memory leak
- Accessing `free`d memory
- Double-`free`ing a pointer
- Exhausting the heap and failing to notice when `malloc` returns `NULL`

Functions often operate in the following way:

1. Acquire heap memory by making calls to `malloc`, or `calloc`, or `realloc`.
2. Perform some operations on the dynamic objects bound to the memory.
3. Return the previously acquired memory back to the heap.

A typical example of using pointers in this way is the use of `malloc` and `free` to create and destroy an object:

```
typedef struct {
  // the particular members of this structure are irrelavant
  char str[256];
} some_struct;

void foo(void) {
  // create an object explicitly ...
  some_struct *p = malloc(sizeof(some_struct));

  // perform some operations on object *p

  // now, dynamically deallocate heap memory
  free(p);
}
```

This function and others similar to it are a source of trouble.

Leaked or orphaned memory

People use `malloc` and then forget to call `free` to return the acquired memory. The program is then said to have a *memory leak* which is a serious problem. Consider a "one-shot" program that acquires memory once, performs some actions on the acquired memory, forgets to return the memory, and then quits. In most cases, it is pointless to deallocate memory just before exiting the program - the operating system will reclaim the memory anyway.

Now, consider a program that must run an indeterminate amount of time. This program intermittently requires copies of a `some_struct` object:

```c
// Given the address of an object of type some_struct, make a deep
// copy of the object on the heap and return a pointer to the block
// of heap memory containing the deep copy
some_struct *somestruct_copy(some_struct const *rhs) {
  // allocate heap memory for a some_struct object
  some_struct *p = malloc(sizeof(some_struct));
  if (p == NULL) { // sanity check
    return NULL;
  }
  // make copies of members of *rhs
  strcpy(p->str, rhs->str);
  return p;
}
```

`somestruct_copy` makes a deep copy of a `some_struct` object on the heap, but it doesn't deallocate it. This introduces the problem that the caller of function `somestruct_copy` must take ownership of the block of heap memory that was allocated by `somestruct_copy`. If the caller forgets to deallocate the memory, a memory leak will arise. Since the function `somestruct_copy` is intermittently called by the program which has to run for an indeterminate amount of time, there will arise a scenario where the memory leaks will gradually fill the heap until allocation requests cannot be satisfied, and the program crashes.

Premature deletion

People `free` an object that they have some other pointer pointing to and later use that other pointer. The other pointer to the "freed object" no longer points to a valid object [so reading it may give bad results] and may indeed point to memory that has been reused for another object [so writing to it may corrupt an unrelated object].

```c
// very bad code ...
void poor(void) {
  // create an object explicitly on the heap
  some_struct *p1 = malloc(sizeof(some_struct));
  // potential trouble - multiple pointers to the same object
  some_struct *p2 = p1;
  // more trouble - p3 is uninitialized and could be pointing anywhere
  some_struct *p3;

  free(p1); // lots of trouble - p2 now doesn't point to a valid object
  // give false sense of safety by saying p1 doesn't point anywhere
  p1 = NULL;

  // p3 may now point to the memory pointed to by p2
  p3 = malloc(sizeof(some_struct));
  strcpy(p3.str, "xyz");

  // more trouble - update (non-existent) "object" pointed to by p2
  strcpy(p2.str, "abc");
  printf("p3.str: %s\n", p3.str); // problem - // may not print "xyz"

  // other code here - but runtime behavior of function is undefined
}
```

Double deletion

An object is freed twice. Double deletion is a problem because memory managers typically cannot track what code owns a resource. Consider:

```
// very bad code ...
void sloppy(int N) {
  // acquire memory from heap
  some_struct *p = malloc(sizeof(some_struct) * N);

  // ... use "array" of some_struct objects pointed to by p ...

  // return the memory back to heap
  free(p);

  // ...     wait a while ...

  // incorrect logic leads to second call to free
  free(p);
  // program behavior is undefined because sloppy doesn't own freed memory
}
```

By the second call to `free`, the memory pointed to by `p` may have been reallocated for some other use and the allocator may get corrupted. In general, a double deletion is undefined behavior and the results are unpredictable and usually disastrous.

Deleting uninitialized pointers

`free`ing a pointer that was never initialized and, therefore, contains a garbage value is a sure path to chaos. The system may try to recover the memory returned by the pointer, which is likely not associated with the heap.

```
// very bad code ...
void chaos(void) {
  some_struct *p; // define but not initialize pointer
  // no references to p ...
  free(p); // error!!! attempting to free garbage
  // program behavior is too terrible to imagine because chaos() is
  // providing free a pointer that contains some garbage value that
  // may point to some location on the heap (which is bad) or may point
  // to non-heap memory (which is really bad) or may point to
  // some region complete outside the program memory (which is the worst)
  // all of these can lead to truly undefined behavior
}
```

On the other hand, function `free` recognizes the `NULL` value and `free` will do nothing when given a pointer having `NULL` value:

```
void harmless(void) {
  some_struct *p = NULL;
  // no references to p ...
  free(p); // harmless!!!
}
```

> ***Programming tip: Reinforce this rule every time you program - a pointer variable should never be defined without being initialized at the same time.***

The reason people make any or all of these mistakes is typically not maliciousness and often not even simple sloppiness; it is genuinely hard to consistently de-allocate every allocated object in a large program once and at exactly the right point of computation.

Debugging programs that use heap memory

As seen earlier, manipulating heap memory is prone to the same bugs as manipulating stack memory:

- Dereferencing uninitialized pointers
- Dereferencing null pointers
- Reading uninitialized memory
- Off-by-one array subscripting
- Getting hosed by a malformed C-style string that doesn't have a null terminating character

and some new ones:

- Failing to `free` allocated memory causing a memory leak
- Accessing `free`d memory
- Double-`free`ing a pointer
- Exhausting the heap and failing to notice when `malloc` or `calloc` or `realloc` returns a `NULL` value

Memory issues come in two flavors: memory *errors* and memory *leaks*. When a program dynamically allocates memory and forgets to later free it, it creates a *memory leak*. A memory leak generally won't cause a program to misbehave, crash, or give wrong answers. Since memory leaks are not insidious, a memory leak is not an urgent situation but a detail that can be resolved at a later time. A *memory error*, on the other hand, is a red alert. Reading uninitialized memory, writing past the end of a piece of memory, accessing freed memory, and other memory errors are insidious activities with potentially catastrophic consequences.

> ***Memory errors should never be treated casually or ignored. You must always prioritize identifying and fixing memory errors before memory leaks.***

Checking a program with Valgrind

With so many ways of introducing memory bugs into our code, what are we to do? Fortunately, an amazing open-source debugging tool called Valgrind is available. According to Valgrind's website,

> Valgrind will save you hours of debugging time. With Valgrind tools you can automatically detect many memory management and threading bugs. This gives you confidence that your programs are free of many common bugs, some of which would take hours to find manually, or never be found at all. You can find and eliminate bugs before they become a problem.

Installing Valgrind is easy:

```
1 $ sudo apt-get install -y valgrind
```

Consider the following source code:

```c
int main(void) {
  int *pi = (int*) malloc(sizeof(int) * 3);
  pi[0] = 11; pi[1] = 21; pi[2] = 31;
  printf("pi[]: %d | %d | %d\n", pi[0], pi[1], pi[2]);
  free(pi); // no problem: malloc() followed by free()!!!
  return 0;
}
```

> ***Debugging tip: Compile your program using `gcc` or `clang` with `-g` option to include debugging information so that Valgrind's summary reports include exact line numbers.***

Suppose you've compiled and linked the above code [after adding necessary includes] into a program `main.out`. Test the program with Valgrind like this:

```
$ valgrind ./main.out
```

Valgrind will print a summary of `main.out`'s memory usage:

```
1  ==210== HEAP SUMMARY:
2  ==210==     in use at exit: 0 bytes in 0 blocks
3  ==210==   total heap usage: 2 allocs, 2 frees, 1,036 bytes allocated
4  ==210==
5  ==210== All heap blocks were freed -- no leaks are possible
6  ==210==
7  ==210== For lists of detected and suppressed errors, rerun with: -s
8  ==210== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Line 8 is the critical part of the summary and is what you should be aiming for: no memory leaks [line 5] and no memory errors [line 7].

Detecting memory leaks

The following code simulates a memory leak that occurs on a fairly common basis:

```c
int* foo(int N) { return malloc(sizeof(int) * N); }

void boo(void) {
  int *pi = foo(3);
  printf("%d\n", pi[0]);
}

int main(void) {
  printf("Calling boo()\n");
  boo();
  // do other stuff after calling boo
}

```

Function `main` calls `boo` which in turn calls `foo`. Function `foo` returns a pointer to the first element of a dynamically allocated array to `boo`. Function `boo` fails in its responsibility of returning the memory back to the heap. To check for memory leaks, you need to include options `--leak-check=full` and `--show-leak-kinds=all` in the `valgrind` command:

```
$ valgrind --leak-check=full --show-leak-kinds=all ./main.out
```

Here's the report from Valgrind for the program:

```
==216== HEAP SUMMARY:
==216==     in use at exit: 12 bytes in 1 blocks
==216==   total heap usage: 2 allocs, 1 frees, 1,036 bytes allocated
==216==
==216== 12 bytes in 1 blocks are definitely lost in loss record 1 of 1
==216==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==216==    by 0x1091A8: foo (in /mnt/c/Users/pghali/Desktop/test/a.out)
==216==    by 0x1091C0: boo (in /mnt/c/Users/pghali/Desktop/test/a.out)
==216==    by 0x1091F9: main (in /mnt/c/Users/pghali/Desktop/test/a.out)
==216==
==216== LEAK SUMMARY:
==216==    definitely lost: 12 bytes in 1 blocks
==216==    indirectly lost: 0 bytes in 0 blocks
==216==      possibly lost: 0 bytes in 0 blocks
==216==    still reachable: 0 bytes in 0 blocks
==216==         suppressed: 0 bytes in 0 blocks
==216==
==216== Use --track-origins=yes to see where uninitialised values come from
==216== For lists of detected and suppressed errors, rerun with: -s
==216== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 0 from 0)
```

It is easy to determine from the report that there is a memory leak: line 5 in `HEAP SUMMARY` indicates that 12 bytes are definitely lost; line 11 provides a `LEAK SUMMARY` that also indicates that 12 bytes were lost; and finally line 20 provides an `ERROR SUMMARY` indicating an error [although not necessarily a memory leak]. Lines 5 through 9 trace the reversed chain of calls that led to the memory leak: `foo` called function `malloc` [at line 5 of source file containing definition of function `foo`] to allocate 12 bytes; `boo` called `foo`; and `main` called `boo`.

Valgrind categorizes leaks in `LEAK SUMMARY` using these terms:

- *definitely lost*: This is heap memory to which the program no longer has a pointer and therefore cannot be freed at program exit. Valgrind knows that you once had the pointer, but have since lost track of it at some earlier point in the program. This memory is definitely orphaned. Such cases can and should be fixed by the programmer.
- *indirectly lost*: This is heap memory that was never freed to which the only pointers to it also are lost. For example, if you orphan a linked list, the first node would be definitely lost, while subsequent nodes would be indirectly lost. Such cases can and should be fixed by the programmer.
- *possibly lost*: This is heap memory that was never freed to which Valgrind cannot be sure whether there is a pointer or not.
- *still reachable*: This is heap memory that was never `free`d to which the program still has a pointer at exit [typically this means a global variable points to it]. Such cases can and should be fixed by the programmer.
- *suppressed*: These are memory leaks potentially caused by libraries. Since programmers can't fix such leaks, they don't want to see them, and therefore Valgrind suppresses such leaks. Suppressed leaks are not a programmer's concern.

Detecting memory errors

While memory leaks are benign and don't cause undefined program behavior, memory errors are insidious and will cause undefined program behavior. Let's look at the list of programmer actions that are liable to cause memory errors: read from an uninitialized variable, dereferencing uninitialized pointers to read from or write to the heap object; dereferencing a `NULL` pointer; reading uninitialized heap memory; off-by-one array subscripting; dealing with a C-style string that doesn't have the null terminator; accessing `free`d memory; double-`free`ing a pointer. Memory errors are insidious and truly evil because a program with such an error seems to work correctly because you manage to get *lucky* much of the time. After several successful such lucky outcomes, you incorrectly feel confident that your program is definitely correct. Later if the program generates a catastrophic outcome, you'll tend to chalk it down to incorrect input or maybe an overheated computer. You might even release the software to a single customer or to millions of users.

> *"...the results are undefined, and we all know what "undefined" means: it means it works during development, it works during testing, and it blows up in your most important customers' faces." -- Scott Meyers.*

Depending on luck is not a good strategy for developing robust software. Using Valgrind can help you track down the cause of visible memory errors as well as find errors lurking beneath the surface that you don't know about. The following subsections show examples of the most common error messages from Valgrind.

Invalid `free()` / `realloc()`

Consider this code that simulates a double-`free`ing:

```
1  int main(void) {
2    int *pi = (int *) malloc(sizeof(int) * 3);
3    pi[0] = 1; pi[1] = 2; pi[2] = 3;
4    printf("%d\n", pi[2]);
5    free(pi);
6    int *pj = (int *) malloc(sizeof(int) * 3);
7    pj[0] = 11; pj[1] = 22; pj[2] = 33;
8    free(pi); // double-deletion
9    printf("%d\n", pj[2]);
10   free(pj);
11   return 0;
12 }
```

Neither `gcc` nor `clang` report any errors even with the full suite of warning options turned on. Nonetheless, Valgrind comes to the programmer's rescue:

```
1  ==222== Invalid free() / delete / delete[] / realloc()
2  ==222==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-
   gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
3  ==222==    by 0x109289: main (in /mnt/c/Users/pghali/Desktop/test/a.out)
4  ==222==  Address 0x4a48040 is 0 bytes inside a block of size 12 free'd
5  ==222==    at 0x483CA3F: free (in /usr/lib/x86_64-linux-
   gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
6  ==222==    by 0x109249: main (in /mnt/c/Users/pghali/Desktop/test/a.out)
7  ==222==  Block was alloc'd at
```

```
==222==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-
gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==222==    by 0x1091F6: main (in /mnt/c/Users/pghali/Desktop/test/a.out)
==222==
33
==222==
==222== HEAP SUMMARY:
==222==     in use at exit: 0 bytes in 0 blocks
==222==   total heap usage: 3 allocs, 4 frees, 1,048 bytes allocated
==222==
==222== All heap blocks were freed -- no leaks are possible
==222==
==222== For lists of detected and suppressed errors, rerun with: -s
==222== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Valgrind reports on line 1 that there is a re-`free`ing in function `main`. This error message is also produced when you try to free a memory block that was not returned by a heap allocator.

Use of uninitialized value and Conditional jump or move depends on uninitialized value(s)

These error messages are generated by Valgrind when uninitialized memory is referenced. Consider code that reads uninitialized memory that was previously allocated by either function `malloc` or `calloc` or `realloc`:

```
int main(void) {
  int *pi = (int*) malloc(sizeof(int)); // pi is uninitialized
  printf("%d\n", *pi);
  free(pi);
  return 0;
}
```

Valgrind provides this abbreviated report:

```
==228== Conditional jump or move depends on uninitialised value(s)
==228==    at 0x48CDAD8: __vfprintf_internal (vfprintf-internal.c:1687)
==228==    by 0x48B7EBE: printf (printf.c:33)
==228==    by 0x109213: main (in /mnt/c/Users/pghali/Desktop/test/a.out)
==228==
==228== Use of uninitialised value of size 8
==228==    at 0x48B181B: _itoa_word (_itoa.c:179)
==228==    by 0x48CD6F4: __vfprintf_internal (vfprintf-internal.c:1687)
==228==    by 0x48B7EBE: printf (printf.c:33)
==228==    by 0x109213: main (in /mnt/c/Users/pghali/Desktop/test/a.out)
==228== HEAP SUMMARY:
==228==     in use at exit: 0 bytes in 0 blocks
==228==   total heap usage: 2 allocs, 2 frees, 1,028 bytes allocated
==228==
==228== All heap blocks were freed -- no leaks are possible
==228==
==228== Use --track-origins=yes to see where uninitialised values come from
==228== For lists of detected and suppressed errors, rerun with: -s
==228== ERROR SUMMARY: 5 errors from 5 contexts (suppressed: 0 from 0)
```

Next, consider code that is a bit different from the previous example - this code fragment contains references to uninitialized variables:

```
int main(void) {
  int ctr; // uninitialized variable - will contain garbage!!!
  for (int i = 0; i < ctr; ++i) {
    printf("%d\n", ctr);
  }
  return 0;
}
```

References to uninitialized variables are reported by both `gcc` and `clang`. Let's look at the abbreviated report from Valgrind:

```
==236== Conditional jump or move depends on uninitialised value(s)
==236==    at 0x1091F6: main (in /mnt/c/Users/pghali/Desktop/test/a.out)
==236==
==236== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

The message on line 1 indicates that the program contains references to uninitialized variables.

Invalid read of size n

Consider code that reads outside the bounds of allocated memory:

```
int main(void) {
  int *pi = (int*) malloc(sizeof(int) * 3);
  printf("%d\n", pi[30]);
  free(pi);
  return 0;
}
```

Neither `gcc` nor `clang` report any errors even with the full suite of warning options turned on. Valgrind again comes to the programmer's rescue:

```
==242== Invalid read of size 4
==242==    at 0x109203: main (in /mnt/c/Users/pghali/Desktop/test/a.out)
==242==  Address 0x4a480b8 is 40 bytes inside an unallocated block of size
4,194,128 in arena "client"
==242==
==242== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Invalid write of size n

Consider code that writes outside the bounds of allocated memory:

```
int main(void) {
  int *pi = (int*) malloc(sizeof(int) * 3);
  pi[30] = 5;
  free(pi);
  return 0;
}
```

Neither `gcc` nor `clang` report any errors even with the full suite of warning options turned on. Valgrind detects the out-of-bounds write:

```
==248== Invalid write of size 4
==248==    at 0x109203: main (in /mnt/c/Users/pghali/Desktop/test/a.out)
==248==  Address 0x4a480b8 is 40 bytes inside an unallocated block of size
4,194,128 in arena "client"
==248==
==248== For lists of detected and suppressed errors, rerun with: -s
==248== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Final words

The entire list of Valgrind error messages are listed here. Remember, memory errors are insidious and can cause a variety of problems ranging from incorrect output and intermittent crashes. Use the code examples in this document to gain experience in detecting and resolving memory errors by running Valgrind. And, make sure to frequently use Valgrind to test your programs in this course [and hopefully beyond] to avoid significant grading deductions.