steps to do:

```
int main(){
        long x = cs1010_read_long();
        //step 1
        long *array = malloc(x *sizeof(long));
        //step 2
        if (array == NULL){
                return 1;
        }else{
                //Use array
        }
        //step 3
        free(array)
}
```

Unit 18: Heap

Learning Objectives

After completing this unit, students should:

- understand the differences between the stack and the heap
- be able to use `malloc` and `calloc` to declare dynamically-sized arrays on the heap
- understand the importance of (i) checking for NULL and (ii) free the memory allocated on the heap
- aware of the possibility of memory leakage and avoid common mistakes that could lead to memory leaks

Auto Variables

We have already seen what a call stack is and how the call stack works in Unit 13. There is another important area of memory used by our programs, called the *heap*.

Recall that a variable allocated on the stack has two properties:

- Its lifetime is the same as the lifetime of the function the variable is declared in.
- The memory allocation and deallocation are automatic.

For stack, the memory is allocated automatically when the function is called and deallocated automatically as soon as the function exits. For this reason, such a variable is sometimes called *automatic* variable, or *auto* variable for short.

Variables on Heap

The memory allocation on the heap can be done automatically or manually. For variables allocated on the heap, its lifetime is either the same as the lifetime of the *whole* program. An example of such a variable is a *global* variable -- a variable that is declared *outside* of any function and can be read or write anywhere in the program. We have banned the use of global variables in CS1010. Using a global variable makes your code hard to understand or reason about:

```
x = 1;
foo();
```

```
2  // { x == ?? }
3
```

Suppose `x` is a global variable, we cannot assert anything about the property of `x` after calling `foo`, since `x` can be modified by `foo` or any function it calls, even though we *never pass `x` into `foo`*. This is worse than passing an array as we have seen in Unit 17! Since now, to assert something about `x`, we need to trace through every line the code to how `x` is updated, even if the function does not take in `x` as a parameter.

Manual Memory Allocation / Deallocation

Allocating memory on the heap, however, is useful if we want to allocate an array dynamically, i.e., not knowing what is the size of the array when we write the program. Often, we need an array whose size depends on the input from the user, such as reading a string or reading a sequence of numbers. We cannot use fixed-length array unless we know for sure that the input size is limited, and we cannot use a variable-length array, since we may get a segfault if the array size is too big for the stack. The only viable solution is to allocate the array on the heap.

man page

The C standard library provides a few functions related to memory allocation on the heap. The header file for these functions is `stdlib.h`. We are interested in `malloc` and `calloc`. `malloc` (memory allocation) is declared as:

```
1  void *malloc(size_t size);
```

It takes in a parameter, `size`, which is the number of bytes of memory to be allocated and returns a pointer to the memory allocated if successful, or `NULL` otherwise. This is a general function so the type of pointer returned is `void *` rather than a pointer to a specific type. The type of `size` is `size_t`, which is a type defined in `stdlib.h` to represent the number of bytes in memory.

The function `calloc` (clear allocation) is declared as:

```
1  void *calloc(size_t count, size_t size);
```

`calloc` allocates memory for `count` items, each of `size` number of bytes, in a contiguous region in the memory and initializes all bits in this memory region to 0. Except for the fact that `calloc` initializes the bits to 0, `calloc(count, size)` is the same as `malloc(count * size)`.

We have seen in Unit 5 that the number of bytes needed to represent a type depends on the platform. Suppose we want to allocate enough memory for, say, 10 `long` values, how

do we know how many bytes are needed? A `long` can be 4 bytes on some platforms, 8 bytes on others. For this purpose, C provides a `sizeof` operator, that returns the number of bytes needed for a type or a variable. So, to allocate memory for 10 `long` values, we say:

need typecast tho

```
1  long *array = malloc(10 * sizeof(long));
```

> **Casting to size_t**
>
> We need to cast `long` arguments into `malloc` and `calloc` as `size_t` to suppress warnings from `clang`.

The CS1010 I/O library, internally, allocates memory on the heap so that we can read in words, lines, or arrays of arbitrary length.

The following shows the example of how `cs1010_read_long_array` is implemented with `calloc`.

```
1  long* cs1010_read_long_array(long how_many)
2  {
3    long *buffer = calloc((size_t)how_many, sizeof(long));
4    if (buffer == NULL) {
5      return NULL;
6    }
7
8    for (long i = 0; i < how_many; i += 1) {
9      buffer[i] = cs1010_read_long();
10   }
11
12   return buffer;
13 }
```

malloc and calloc will not always allocate

so need to check if memory has been allocated

then free everything at the end(in main)

> **Casting to size_t**
>
> We need to cast `long` arguments into `malloc` and `calloc` as `size_t` to suppress warnings from `clang`.

Memory Deallocation

Even though the memory available on the heap is larger than the stack, it is not unlimited, and therefore we should still use the memory judiciously. In particular, after we are done using the memory allocated to us, we should call the method `free`, passing in the pointer the memory region allocated, to have the memory region deallocated, returned to the OS to be reused by others.

A common bug is for a programmer to access a memory that has been freed. This would cause strange behaviors, possible crashes in random places since we will be accessing memory that is being used by others. It is a good practice to set the pointer to NULL after `free`-ing the memory region so that we do not accidentally use it.

```
free(buffer);
buffer = NULL;
```

Another common bug is for programmers to request memory via `malloc` or related functions, but forgot to `free` it back to the OS. As a result, as the program runs, it starts to hog the memory and the system will become slower and slower. This bug is known as *memory leak*.

Another possible bug is for programmers to change the pointer to the region of memory allocated. For instance,

NOTE

```
long *buffer = calloc((size_t)how_many, sizeof(long));
buffer = cs1010_read_long_array(how_many);
```

After we execute the code such as the above, the pointer `buffer` will point to something new, and there is no longer a pointer pointing to allocate memory. The memory allocated becomes unreachable, and therefore we can no longer free it!

In CS1010, from now on, you are to make sure that memory that is allocated via `malloc` and related functions are `free` after it is used, including those allocated in the CS1010 I/O library. The API documentation tells you what are the values returned by the library that should be deallocated by the caller via `free`.

Problem Set

Problem Set 18.1

Draw the call stack and the heap, showing what happened when we run the following code:

```
void foo(long *y, long *z)
{
  y[0] = -7;
  y[1] = -8;
  z[0] = 4;
  z[1] = 5;
}


int main()
```

```
11  {
12    long y[2] = {1, 2};
13    long *z = calloc(2, sizeof(long));
14
15    z[0] = y[0];
16    z[1] = y[1];
17
18    foo(y, z);
19  }
```

Problem Set 18.2

Read the man page for the function `realloc` and explain what does it do. Can you come up with a situation where it could be useful?