SAXION UNIVERSITY OF APPLIED SCIENCES

Algorithms & Data Structures

School of Life Sciences, Engineering & Design

Topic:	**Memory, pointers, the stack & and the heap**	Lesson:	**1**
Lecturers:	*Robert de Groote* *Dawid Zalewski*	Revision:	**February 1, 2022**

(C has) the power of assembly language and the convenience of ... assembly language.
Dennis Ritchie (creator of C)

Learning Objectives

After successfully completing this activity a student will:

- Be able to explain what the lifetime of an object is.
- Understand the differences between the stack and the heap.
- Be able to use `malloc`, `realloc` and `free` to manage dynamic memory in C.

Team

Before you start this activity, decide on your team members' roles and fill in the table below. Combine the roles of *Spokesperson* and *Reflector* in groups of three.

Team:	**Date**:
Facilitator keeps track of time, assigns tasks and makes sure all the group members are heard and that decisions are agreed upon.	
Spokesperson communicates group's questions and problems to the teacher and talks to other teams; presents the group's findings.	
Reflector observes and assesses the interactions and performance among team members. Provides positive feedback and intervenes with suggestions to improve groups' processes.	
Recorder guides consensus building in the group by recording answers to questions. Collects important information and data.	

Activities

In this week, you will begin creating your first data structure in C - an array that has a capacity that can be changed during the runtime of a program.

To be able to implement such a dynamic array, you will first need to revisit *pointers*. In addition, you will have to build up an understanding of the *lifetime* of objects, and learn about different memory segments where objects are stored - the *heap* and the *stack*.

You will learn the necessary details by working, together with your teammates, on various activities. In your team, there are different roles. Before you begin, discuss who will take on which role. Make sure to switch roles every week.

After completing the activities of this week, the *Spokesperson* of the group will present the group's findings. The presentation is meant to be a summary of the following aspects:

- the things you've learned, *in your own words*,
- the things that were the most difficult to grasp,
- the process of finding the right information and applying it,
- how you worked together as a group,
- what you would differently or similarly in the future.

Therefore, make sure to keep a log of your findings (this is the *Recorder*'s task) - the answers to the questions, the implemented code, results, etc.

Arrays

One of the most basic data structures that is available in the C programming language, is the *array*. Arrays are used to store sequences of values of the same type, such as daily air temperatures in a simple weather tracking system, or the names and birthdays of persons in a birthday calendar:

```
1  typedef struct {
2    int year, month, day;
3  } date_t;
4
5  typedef struct {
6    const char * name;
7    date_t birthday;
8  } person_t;
9
10 #define CAPACITY (100)
11
12 int main( void ) {
13   person_t persons[CAPACITY];
14 }
```

When declaring an array such as `persons` in the code fragment shown here, you have to choose a capacity - the amount of objects that can be stored in the array. Here, `persons` is declared as an array able to hold 100 objects of type `person_t`. We used a *pre-processor* defined constant `CAPACITY` with a value of 100, instead of directly putting a number in the `persons` declaration (like `person_t persons[100]`), because it is considered a very bad programming style to use raw number literals in code. Such, out of the blue appearing numbers are called *magic numbers* and they are frowned upon by most programmers.

When you do not know beforehand how many objects will need to be stored (this might, for example, depend on the user's input), choosing a fixed capacity can be difficult.

Selecting a capacity that is overly pessimistic (very high) will mean that your program uses a lot of memory that is not used, which can be a problem especially on embedded systems where resources are scarce. Choosing a capacity that is too low, on the other hand, means that only few values can be stored - and the system is not able to deal with larger amounts of data.

In this week, you'll work your way towards implementing an array that has a *dynamic* capacity. In other words, the capacity of the array can be changed during the runtime of the program. This means that there is no need to decide on a fixed capacity beforehand - it can be increased when necessary. This will serve as a basis for understanding the data structures that you will explore during this course.

Variables, memory and lifetime

Before we start, let's define some terms that we'll use frequently: *object*, *variable*, *value* and *identifier*. You most likely heard all of them, quite possibly used exchangeably. In the following line of code:

```
1 int number = 42;
```

We have an *object* of type `int`, whose *value* is `42`. We refer to this object through the *identifier* `number`. Consequently,

- *Object* is a chunk of computer *memory* with a given *type*.
- *Value* is the byte sequence stored in an *object* (a binary representation). Depending on the *type* of an *object*, the same byte sequence can be interpreted differently. The bytes `{0x2a, 0x00, 0x00, 0x00}` are interpreted as `42` for an `int` object (that's our `number`) and as a string `"*"` if it was a *string literal* (`const char*`) we were talking about.
- *Identifier* is a name, used in a program, through which we can refer to an *object*.

That leaves us with *variable* - that's the most vaguely defined term. When we talk about *variables* we usually mean a combination of an *object* and its *identifier*. So let's keep it like this, a *variable* is both an *identifier* and the *object* it refers to (or simply an *object* that has a *name*).

All programs need objects - without objects, no data can be stored or processed. For obvious reasons, objects usually have identifiers attached to them, so we can simply talk about variables. The programs that you have written in the previous programming courses probably already contain quite a number of variables, despite their relative simplicity. Complex programs like the kernel of an operating system or the control system of an industrial plant may contain hundreds of thousands to millions of objects and nearly as many identifiers.

All variables are stored in memory, but most of the variables are used only for a short while. In this section, you will write many small programs that let you experiment with variables, and the way they are stored in memory.

In order to find out at which memory location (or *address*) a variable is stored, you can use the debugger. Alternatively, you could print the memory addresses of the variables of your interest, using the `printf` function and the `%p` format specifier.

Memory

How much memory do you need to store 10 objects? And how much for 20 objects? This of course depends on the size of each object, and the size is determined by its *data type* . In C, there are several *fundamental* data types - `int`, `float`, `double`, `char`, etc[1]. In addition, *arrays* and *structures* can be used

[1] types can also be declared to be unsigned or signed, but that has no impact on the size of the type, only on its interpretation.

as containers to group together multiple related objects. Finally, *pointer* types hold *memory addresses* of objects of a given type, or of unknown type (indicated with `void *`).

Each data type occupies a predefined number of bytes in memory, used to store *values* of that type. Objects of the type `char`, for example, use only a single byte, whereas a `double` object takes up 8 bytes. The purpose of the next exercise is to refresh your knowledge about data type sizes in C, and the `sizeof` operator.

Activity 1: Memory usage - the sizeof operator

Complete the table given below by writing a program that uses the `sizeof` operator to determine and print the sizes (number of bytes) of the listed data types. Note that the value returned by the `sizeof` operator is `size_t`, which is equivalent to an *unsigned* integral type (`unsigned int`, `unsigned long`, etc.). To print the result of the `sizeof` operator, use either the `\%u` or `\%lu` format specifier.

- Fill out the table below
- Run the program you've used to obtain the data type sizes in the online compiler explorer. Do you see any differences in the sizes reported?
- Does the size of a pointer depend on its type? Why (not)?

Data type	Size in bytes
`char`	
`short int`	
`int`	
`long int`	
`long long int`	
`float`	
`double`	
`long double`	

Pointer data type	Size in bytes
`char*`	
`int*`	
`float*`	
`double*`	
`void*`	

Apart from the basic data types that you've analysed in the previous activity, there are two basic *aggregate* data types in C - *arrays* and *structures*. The `sizeof` operator can be used to find out how much memory variables of these types occupy. In the following activity, you will explore the memory usage of these two data types.

Activity 2: Array and structure sizes

In the code fragment shown below, the size of an array and a `struct` variable is printed twice. Once directly as the size of a local variable, and once as the size of an argument passed to a function. Run the program on your machine. Do you observe any differences? Explain the output that is printed in the console.

```
1  typedef struct {
2    char name[80];
3    int age;
4  } person_t;
5
6  void size_array(int array[]) {
7    printf("Size of array parameter: %lu\n", sizeof(array));
```

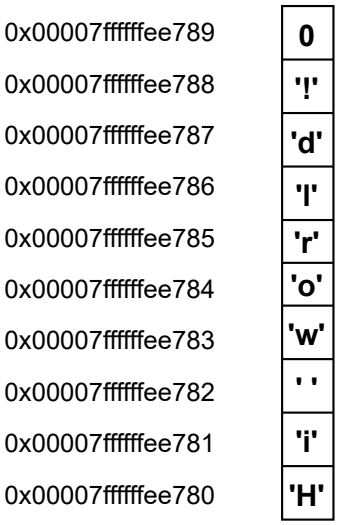
↪ *Activity continues on the next page.*

```
8     }
9
10    void size_struct(person_t p) {
11      printf("Size of p parameter: %lu\n", sizeof(p));
12    }
13
14    int main() {
15      int array[10];
16      person_t bob = {.name = "Bob", .age = 22};
17
18      printf("Size of int array: %lu\n", sizeof(array));
19      printf("Size of person_t structure: %lu\n", sizeof(bob));
20
21      size_array(array);
22      size_struct(bob);
23    }
```

Now that you know more about the sizes of different data types, it's time to look at how and where variables of these types are stored in memory. As a simple example, consider the following basic program:

```
1   int main( void ) {
2     char string[] = "Hi world!";
3     printf("%s\n", string);
4   }
```

The variable `string` is a local variable that is stored in memory as follows (the actual memory addresses may differ across platforms / compilers):



The exact addresses on your machine can be printed using a loop:

```
1   int main( void ) {
2     char string[] = "Hi, world!";
3
4     for (size_t i=0; i < sizeof(string)/sizeof(string[0]); ++i){
5       printf("%c: %p\n", string[i], (void*)&string[i]);
```

```
6         }
7       }
```

In the following activity, you will analyse how other variables are stored in memory.

Activity 3: Memory addresses

The program listed below prints the memory addresses of several variables. Run the following program in CLion and debug the program to find out which variables occupy which memory addresses.
Create a table similar to the one shown earlier (containing the 'Hi world!' string), which shows how the variables are stored in memory and what the exact contents of the memory are[a].

```
1   int main( void ) {
2     int a = 0xA0B0C0D0;
3     short int b = 0x7856;
4     const char * s = "Hello!";
5     char buf[] = "Pointer";
6     short int c = 0x3412;
7
8     printf("int a: %p\n", (void*) &a);
9     printf("short int b: %p\n", (void*) &b);
10    printf("const char *s: %p\n", (void*) &s);
11    printf("char buf[]: %p\n", (void*) buf);
12    printf("short int c: %p\n", (void*) &c);
13  }
```

Next, do the following:

- Add a `printf` statement to print the memory address of the **first character** of the "string" `s`. Are the characters of the string "Hello" stored near the variables `a`, `b`, `s`, and `c`?
- Are the variables layed out in a contiguous way in memory, or are there gaps between them?

[a] You can use an online table generator to generate a table that can be pasted in your log.

The Stack

If every single variable of each function in a program would be stored in a separate memory location, then a program would require lots of memory. Fortunately, only a few variables are *alive* at any point during the execution of a program. These *live* variables are the only variables that must exist in memory (at that given point during program execution).

To see which variables are *alive*, you can use the debugger in CLion, and explore the different *stack frames* that are available as the program executes. Another way to inspect variables and their memory addresses is to simply put a few `printf` statements in your program, and have them print the addresses of the variables of interest. This is what you'll do in the following two activities.

Activity 4: Observing automatic lifetime

In the program listed below, add `printf` statements to the `add` and `mul` functions, to print the memory addresses of the local variables `a`, `b`, `c`, `x`, `y`, and `z`.

Make sure that your compiler is set not perform any code optimization. This could be the default setting for your IDE, but in case you're unsure, the compiler flag you'll need is `-O0`.

When running the program, are the memory addresses of any of these local variables reused by

↪ Activity continues on the next page.

other local variables? Why (not)?

```
int add(int a, int b) {
  int c = a + b;
  return c;
}

int mul(int x, int y) {
  int z = x * y;
  return z;
}

int main( void ) {
  printf("%d\n", mul(add(3, 4), add(1, 5)));
}
```

Two variables can't co-exist at the same memory location - if two variables need to be *alive* at a given point during the execution of a program, then they must be stored at different memory addresses.

By looking at which functions are called by which functions, and in which order, you can predict[2] whether the variables of different functions will share their memory location, or whether they will be located at different addresses, as the program is being executed. Try to make this prediction yourself during the following activity.

Activity 5: Observing the stack

In the program listed below, add `printf` statements to the `bar` and `foo` functions, to print the memory addresses of the local variables `a`, `b`, `x`, `y`, `bx`, and `by`.

When running the program, are the memory addresses of any of these local variables reused by other local variables? Why (not)?

```
int poly(int a) {
  int b = a * (a + 1);
  return b / 2;
}

int add_polys(int x, int y) {
  int bx = poly(x);
  int by = poly(y);
  return bx + by;
}

int main( void ) {
  printf("%d\n", add_polys(42, 24));
}
```

At this moment, a pattern should become apparent: compilers reuse memory locations for variables that are declared locally within functions. This is just one of the observable consequences of the existence of the *stack*.

[2] Predictions are straightforward when compiling the code without any optimizations (using the -O0 flag), but don't try this when compiling with full optimization!

The *stack* is a block of a program's memory where all *local variables* reside. When the control flow enters a function, all the variables declared in this functions (including its parameters) are *allocated* in the order of declaration on the stack, one after another. When the control flow leaves the function, the stack space used by its variables is reclaimed. Consequently, those objects cease to exist. We say that their *lifetime* has ended.

The *lifetime* of variables is bound to the *scope* in which they are declared. In C, a new scope opens:

- Whenever a function is called.
- Whenever a new code block is introduced with curly braces.
- Whenever a loop is executed. All variables declared within a loop belong to its scope.

Variables that are declared within a scope live (on the stack) only as long as the control flow is within this scope - their *lifetime* is bound to the scope. Such scope-bound, stack variables are called variables with *automatic lifetime*. That's because their existence is automatically managed by a compiler that puts them on the stack and removes them from it, reclaiming space.

Variables with automatic lifetime are easy to use but they don't come without potential dangers. One of the common bugs happens when a memory address of a local variable declared within a function *leaks* to the surrounding scope. Usually, this is caused by a careless programmer returning a pointer to such a local variable, like in the following activity.

Activity 6: Leaking local addresses

In the program given below, `get_answer` returns a memory address (a pointer) of a local `int` variable. This pointer is captured in the `main` function and assigned to `ptr_answer`. Then, the integer value pointed by it is printed to the standard output. In the next line, the function `i_do_nothing` is called - this function, as its name suggests, does absolutely nothing useful, it just declares a local array of four integers. Finally, the value pointed to by `ptr_answer` is printed once again.
Compile this program with the `-O0` optimization level and run. What do you observe? Explain why this happens.

```
1  int* get_answer( void ){
2    int answer = 42;
3    int* ptr_answer = &answer;
4    return ptr_answer;
5  }
6
7  void i_do_nothing( void ){
8    int no_answer[] = {24, 24, 24, 24};
9  }
10
11 int main( void ) {
12   int* ptr_answer = get_answer();
13   printf("The answer is: %d\n", *ptr_answer);
14
15   i_do_nothing();
16   printf("The answer is: %d\n", *ptr_answer);
17 }
```

Dynamic memory allocation

All (non-static) variables that are local to a function (note that this includes the function's parameters!) have a lifetime that is bound to their scope - this is called *automatic* lifetime. These variables are stored on the stack, and the part of the stack they are stored in is reclaimed when they go out of scope - typically this happens when the function they are declared in ends.

This is not always desired. For example, suppose that in a function you would like to *create* an array that can subsequently be used by other parts of the program for storing data. In other words, it should still exist after the function ends.

Using local variables, the only way to achieve this is by creating this array in the `main` function, so that it's lifetime is bound to the runtime of the program. Alternatively, you could introduce global variables.

However, that won't help you if you need multiple arrays and you don't know upfront exactly how many. This would also mean that you'd need to set up all the arrays (and other data structures) giving them fixed properties (like the capacity).

There's an obvious need to be able to make room for data on request, during the runtime of the program. In the next activity, you'll make a first attempt to write a function that should fulfil exactly that need.

Activity 7: Memory addresss of local variables

The following program is an attempt to create an array in a function.
The `do_some_work` function is not specified - you can fill in the details yourself. Compile and run this program with the maximum warnings levels set: `-Wall -Wextra -pedantic`.

- Which warnings and / or errors does the compiler give when compiling this program?
- What do these warnings and / or errors mean?
- Does this approach to create an array at runtime work? Why (not)?

```
1  void do_some_work(int *values, int count);
2
3  int * create_int_array() {
4    int array[10];
5    return &array[0];
6  }
7
8  int main( void ) {
9    int * array = create_int_array();
10   for (int i = 0; i < 10; i++) {
11     array[i] = i + 1;
12   }
13   do_some_work(array, 10);
14 }
```

Besides the stack, which is used for objects with automatic lifetime, data can also be stored in a different part of memory - the *heap*. The lifetime of objects stored on the heap is fully controlled by a programmer. By calling special library functions, memory can be reserved on the heap to make storage for new objects, or released, if it's not needed any longer. Memory *allocation* (reserving) and *freeing* (releasing) does not necessarily happen in the same function or even in the same source file. Quite to the contrary, it's common to *allocate* memory on the heap in one part of a program and *free* it in some distant, unrelated place. This kind of manual memory management is called *dynamic* and objects that use *dynamic memory* for storage are said to have *dynamic lifetime*.

Functions that are used to reserve and release heap memory are declared in the `stdlib.h` header. *Allocating* memory dynamically is done with the `malloc` function (Memory-ALLOCate), while *freeing*

such memory is the task of the `free` function.

In the upcoming activities, you'll learn more about how to dynamically allocate memory and release it again.

Allocating memory

To store data on the heap, memory can be allocated using the `malloc` function. This function takes the number of bytes to allocate and returns the *memory address* of the first byte of the allocated memory block if it succeeds (it returns 0 (`NULL`) if it does not succeed). The type of an object that is going to occupy memory reserved with `malloc` is undefined. To reflect this, the memory address that is returned by `malloc` has the type `void *`. `void` is a so-called *incomplete type* - used when there's insufficient information to determine the exact type. A compiler cannot possibly guess what kind of object we want to store in the allocated memory, but we usually know it very well. That's why the memory address returned by `malloc` is cast to a pointer of type that we want to work with.

For example, the following code allocates 1 mebibyte of memory and stores the address of the first byte in this memory block in the variable `ptr`. Since this dynamically allocated memory is intended to be used for storing 131'072 `double` numbers, the address returned by `malloc` is cast to `double *`:

```
1 double *ptr;
2 ptr = (double*) malloc(1024 * 1024);
```

Instead of using a number of bytes to specify how much memory must be allocated, the size of the memory block is typically expressed in terms of two things: the size of the *data type* that is to be stored in a block of memory, and the *number of elements* that need to be stored in it.

As you've seen before, the data type size can be obtained using the `sizeof` operator. In the following activity, you'll use `malloc` to allocate memory for all kinds of data type storage.

Activity 8: Using malloc

Write a program (include the program in your log) that uses the `malloc` function to:

- allocate memory to store one `unsigned long` number,
- allocate memory to store 256 `float` numbers,
- write a function that takes a `count` as its argument, allocates enough memory to hold `count` `int`s and returns the pointer to the allocated memory block:
  `int* allocate_memory(int count);`

There are many ways of specifying the size of multiple objects of some type, try at least the following two:

- `sizeof(Type)* count_of_objects` (e.g. `sizeof(long)* 100`)
- `sizeof(Type[count])` (e.g. `sizeof(long[100])`)

You've already seen how array variables, when used in expressions, are treated like pointers to the first element of a corresponding array. Consequently, the following function signatures are equivalent:

```
1 void print_stats(long primes[], int size);
2
3 void print_stats(long *primes, int size);
```

In both cases, `primes` is just a pointer to a value of type `long`, which also happens to be the first item of an array of `long`s with `size` elements.

Declaration syntax aside, the array-pointer duality extends much further, for instance, the two memory addresses printed out by the program listed below are identical:

```c
int main( void ) {
  double * ptr = (double*) malloc(10 * sizeof(double));
  printf("Memory address stored in ptr: %p\n", (void*) ptr);
  printf("Memory address of ptr[0]: %p\n", (void*) &ptr[0]);
}
```

In a similar way, all three functions shown below perform exactly the same computations:

```c
// uses array syntax
void print_stats(int size, long primes[]){
  long min = primes[0], max = primes[0];
  long sum = 0;

  for (int i=0; i<size; ++i){
    sum += primes[i];
    if (min > primes[i]) min = primes[i];
    if (max < primes[i]) max = primes[i];
  }

  printf("min = %ld\nmax = %ld\nmean = %0.1f", min, max, sum/(double)size);
}

// uses pointer syntax that closely mimics array syntax
void print_stats(int size, long primes[]){
  long *min = primes, *max = primes;
  long sum = 0;

  for (int i=0; i<size; ++i){
    sum += *(primes + i);
    if (*min > *(primes + i)) min = primes + i;
    if (*max < *(primes + i)) max = primes + i;
  }

  printf("min = %ld\nmax = %ld\nmean = %0.1f", *min, *max, sum/(double)size);
}

// uses only pointers, departing from array-inspired code
void print_stats(int size, long primes[]){
  long *min = primes, *max = primes;
  long sum = 0;

  for(long *current = primes; current != primes + size; ++current){
    sum += *current;
    if (*min > *current) min = current;
    if (*max < *current) max = current;
  }

  printf("min = %ld\nmax = %ld\nmean = %0.1f", *min, *max, sum/(double)size);
}
```

In a way, every pointer can be interpreted as a memory address of the first element of an array consisting of contiguous objects of the same type. Sometimes, it just happens that there's only one element in such an array and then the pointer points to a single value. When using `malloc`, we always allocate a contiguous block of memory that can, depending on its size, host multiple objects. In the previous

activity you used this property to allocate memory that could hold 256 `float` numbers or `count` `int` numbers, where `count` was a function parameter. In either case, you were, in fact, allocating an array of objects. Now, you'll go a step further, making a usable function that allocates and returns a *dynamic array*.

Activity 9: Using allocated memory as an array

The program listed below is a failed attempt to allocate memory for storing an array of 20 `int`egers. Answer the following questions:

- How many `int` elements can be stored in the allocated block of memory?
- What happens when you perform an out-of-bounds access to an array that is stored in dynamically allocated memory?
- What is the problem in the program listed below, and how can it be fixed?

```
int main( void ) {
  const int capacity = 20;
  int *ptr = (int*) malloc(capacity);

  for (int i = 0; i < capacity; i++) {
    printf("ptr[%d] = %d\n", i, ptr[i]);
  }
}
```

How much memory can be allocated by a program? This of course is very much dependent on the hardware architecture and operating system that the program is running on. Embedded devices typically have at most only a few megabytes of memory, whereas desktop or laptop computers have tens of gigabytes. When no memory can be allocated, the `malloc` function returns a special value: `NULL`[3]. To find out what the memory usage limitations are on your system, explore the following activity.

Activity 10: Infinite memory?

The program listed below allocates blocks of 2^{28} bytes, or 256 mebibytes.
Why does the program crash, and how many blocks can be allocated before the program crashes? Add `printf` calls to check this.

Try to run this program on a number of different machines, to see if you can get different outcomes.

```
int main( void ) {
  const unsigned int block_size = 1 << 28;
  void *ptr = malloc(block_size);
  while (ptr != NULL) {
    ptr = malloc(block_size);
  }
}
```

Releasing memory

Memory is typically used to temporarily store some data, so that it can be processed. Different kinds of computations and analyses are run on stored data, and as soon as the results are ready, the data itself is usually no longer needed. This means that the memory that was used to store the data can be

[3] `NULL` isn't really such a special value - it's just a preprocessor constant with a value of `0`. In C a memory address of `0` is treated as invalid - it cannot be read from or written to.

used for other purposes. Memory is a limited resource, and programs should release memory once it is no longer needed.

Releasing (or *deallocating*) dynamically allocated memory in C is done by calling the `free` function and passing a pointer to a memory block that was obtained by a previous call to `malloc`:

```
const int SIZE = 512;

char *str = (char*) malloc(sizeof(char[SIZE]));

// do something with str
...
// str is no longer needed: reclaim its memory:
free(str);
```

Activity 11: Fixing a memory leak

Fix the program of the previous activity, by calling `free` at the right place to release the allocated memory. Verify that your program does not crash by letting it allocate (and deallocate) a number of blocks that is twice as high as the number of blocks that caused the previous program to crash.

The `free` function releases (or deallocates) memory, so it can be reused by future calls to `malloc`. But what happens if you try to release memory that was not allocated? You'll try this in the next activity.

Activity 12: Dangerous `free`s

Find out what happens when you try to call the `free` function on a pointer that was not obtained using `malloc`, and on a pointer, whose value is `NULL`:

```
int stack_variable = 42;
int* ptr = &stack_variable;
free(ptr);

int* null_ptr = NULL;
free(null_ptr);
```

As you've learned in the previous activities, dynamic memory allocation requires great care - memory that is allocated must also be released. It is the programmer's responsibility to make sure that no memory is *leaking* - allocated but not released. Fortunately, there are tools available to help the developer track down memory management problems, so that, among others, memory leaks can be fixed. To learn more about these tools, read up on Address Sanitizers.

Reallocating memory

When allocating memory on the heap for an array of a chosen capacity, a natural question is: how to choose the right initial capacity? If you expect having to store many elements, then you could choose a higher capacity than when the expected number of elements is low, but sometimes it's difficult to predict how many elements there will be.

Fortunately, it's possible to adjust the size of a memory block that you've allocated, in case it turns out to be of insufficient size. This involves the following steps:

1. **allocate** a new memory block with more capacity than the original,
2. **copy** all the elements from the *old* storage to the new one,

3. **reclaim** the *old* memory (it is not needed any more),
4. **reassign** the previous pointer to point to the new storage,
5. **update** the known `capacity` to its new value.

By using `malloc` and `free` together, this can be achieved through the following code:

```
#include <string.h> // for memcpy
#include <stdlib.h> // for malloc and free

int main( void ) {
  unsigned int capacity = 10;
  float *grades = malloc(sizeof(float[capacity]));

  // ... fill the grades array with 10 grades

  unsigned int new_capacity = 2 * capacity;
  float *new_grades = malloc(sizeof(float[new_capacity])); // allocate a new block
  memcpy(new_grades, grades, sizeof(float[capacity]));  // copy the contents
  free(grades);    // reclaim the old memory
  grades = new_grades;  // make the new block the current block
  capacity = new_capacity; // make the new capacity the current capacity

  /* ..... add more grades ... */

  free(grades); // reclaim memory
}
```

As you can see, this is all quite involved. In fact, the code is also incorrect - it may be that the second call to `malloc` fails (because there is insufficient memory available), meaning that no copying should take place, canceling the reallocation altogether.

Another thing that the code above can't possibly deal with is the situation in which there's sufficient memory available directly following the current block of memory. In that case, instead of copying the contents of the current block to a fresh block of memory, the current block of memory can simply be expanded, rendering the copying unnecessary.

Fortunately, the `realloc` function is there to easily let you reallocate memory, taking all these details into account. You will learn more about this function in the following activity.

Activity 13: Using realloc

Look up the documentation of the `realloc` function on cppreference.com. How many explicit calls to `free` must be added to the program listed below, in order to release all the memory that has been allocated?

```
int main( void ) {
  float *grades = NULL;
  for (int capacity = 10; capacity <= 100; capacity += 10) {
    float *new_grades = (float*)realloc(grades, sizeof(float[capacity]));
    if (new_grades != NULL){
      grades = new_grades;
    }
  }
}
```

That's it - you know should know enough about memory, the stack, lifetime of variables, and dynamic memory allocation, to apply this in programs yourself. In the following activity, you will use dynamic memory allocation to read the contents of a file into memory, without having the need to specify beforehand how much memory you need. The knowledge that you have obtained by working on these activites will serve as a basis for creating your first data structure in C, which you will do in the next week.

Activity 14: Using a dynamically sized buffer

The program listed below reads a file into memory, which it allocates dynamically. Initially, it allocates only a small amount of memory to hold the file's contents. Therefore, the program will access the memory out of bounds, which crashes the program, after it has read some characters.

Fix the program by *reallocating* a bigger chunk of memory once the `count` of characters stored reaches the current `capacity`. Use a factor of around 1.5 to increase the capacity each time it has been reached. Test the program by creating a file "input.txt", and pasting some contents into it.

```
1  int main( void ) {
2    char *ptr = NULL;
3    unsigned long capacity = 20;
4    ptr = realloc(ptr, sizeof(char[capacity]));
5    if (!ptr){
6      fprintf(stderr, "Memory allocation failed\n");
7      return 1;
8    }
9
10   FILE *file = fopen("input.txt", "r");
11   if (!file) {
12     fprintf(stderr, "Error opening file\n");
13     return 1;
14   }
15
16   unsigned long count = 0;
17   int c = fgetc(file);
18   while (c != EOF) {
19     /* re-allocate memory pointed to by ptr if count == capacity
20  * don't forget to check if the pointer returned by realloc is not NULL
21  */
22     ptr[count++] = (char) c;
23     c = fgetc(file);
24   }
25 }
```