1 Memory

The memory available to a C++ program is split into three segments: the text, the stack, and the heap. The text stores data associated with the compile phase of the build phase, such as static variable, const variables, constexpr variables and functions, global variables, etc. Mostly in this class we will be dealing with the stack and the heap, so these two portions of memory are what we focus on here.

1.1 The stack

When you create a POD (plain old data) variable in C++ using syntax like `int x = 5`, the memory associated with the variable lives in “the stack.” To understand “the stack” in C++, one should first understand the data structure called “a stack.” Think of a stack of plates in a cafeteria: you can only remove the top plate and new plates added to the stack of plates have to be placed at the top. This is how the data structure called a stack works: new data has to be placed at the top of the data structure, and only data at the top can be accessed or removed. The stack in C++ is a stack, but instead of holding single pieces of data like integers, it holds collections of data called “stack frames.” A stack frame is added to the top of the stack whenever a function is called, and the stack frame contains all of the local variables associated with a function (parameters and variables created in the function) as well as the memory locations of variables to be modified by the behavior of the function (the code `int x = fun()` would give the stack frame of the function fun the memory location of x). When a function is finished running, its stack frame is deleted from the stack. Only stack data in the top stack frame can be accessed at any given time in a program.

```cpp
int fun(int a) {
    int b = 5;
    return a*b;
}
int main() {
    int x = 100;
    int z = fun();
}
```

In the above code, a stack frame is added to the stack when the function main() is called, and ints called x and z are stored in the stack frame. When fun(int a) is called, a new stack frame is added to the stack, containing ints a and b as well as the location in memory of the int z in main. When the return statement of fun is run, the value of z in main is modified, and the stack frame associated with fun is removed from the stack. Finally, when main finishes running, its own stack frame is removed and the program exits.

Using the ideas in the first paragraph of this section, one can make the idea of scope more concrete by defining the scope of a block of code to consists of all of the variables in a stack frame. However, it is not the case that every block of code creates a new stack frame. Indeed, when a new block
of code is created, such as by an if-statement or a for loop, the same stack frame is used but it is noted which variables are created within the block of code. This results in the new block of code having access to all of the variables in the current stack frame, which is why an if-statement can access variables created prior to the if-statement. Because it is noted which variables were created within the block of code, the loader is able to delete data local to a block of code when a block of code is finished running.

```c
int main() {
    int x = 1;
    // stack frame contains int x
    if (x == 1) {
        int y = 2;
        // the scope of this block consists of int x and int y
    }
    // int y deleted and no longer exists in current stack frame
}
```

### 1.2 The Heap

When using the stack, the programmer has no control over when memory is deallocated; indeed, a variable local to a block of code is only deallocated when the block of code ends. The designers of C++ wanted to give programmers more control over memory, allowing programmers to explicitly allocated and deallocated memory, so the heap was created. The heap is a central repository of data that is accessible from any part of the program. Memory is allocated using the keyword `new`, deallocated with the keyword `delete`, and the location of data is kept track of using pointers. The heap is much larger than the stack, since the designers of C++ assumed that saving data to the heap would be more popular than saving data to the stack, so large arrays, etc. are often saved on the heap.

One downside of allowing programmers to have great control over memory allocation and deallocation is the possibility of “memory leaks.” As stated in the previous paragraph, pointers are used to keep track of where data is stored in the heap. Pointers, however, are local variables on the stack, and local variables are deleted when their containing block of code ends. Thus, it is possible that a pointer can be deleted unintentionally when a block of code ends, in which case the location of data on the heap would be lost. This is called a memory leak: when the location of heap data is lost. Because the data is still allocated even after a pointer is deleted, the memory associated with the deleted pointer will be allocated for the duration of the program. This is bad, because allocated memory cannot be repurposed for another task, so if we have recurrent memory leaks then eventually it is possible that there will be no memory left to use in the program.

```c
fun() {
    int* arr = new arr[10000]; // create an array of ints of size 10000
}
int main() {
    fun();
    // the array allocated in fun() still exists in the heap, but its location is lost
    // more code
}
```

In the above code, an array is allocated onto the heap in the function `fun()`. When `fun()` finishes running, the variables local to `fun()` are deleted, which includes the pointer `int* arr`. The result is for the duration of the program after `fun` is called, the array will exist on the heap and take up
space, and we will be unable to do anything about it because we do not know where it is located. To fix this, we would modify the code by adding a delete[] line:

```cpp
fun() {
  int* arr = new arr[10000]; // create an array of ints of size 10000
  delete[] arr;
}
```

Memory leaks commonly affect performance in applications like video games. A video game needs to run for possibly hours at a time, during which memory is constantly being allocated as the human player interacts with the game. If memory is not deallocated appropriately, then memory leaks will build up and eventually the game will become very slow as the loader has to spend a long time searching and rearranging memory so that it has enough space. Eventually the game will crash if there is no space to store any new data.

## 2 Arrays and pointers

### 2.1 Using pointers and arrays

In general,

1. Heap data needs to be deleted before the pointer to that data is deleted

2. It is wise to set a pointer to nullptr after its data is deleted. This is because the following facts allow us to do useful things
   
   (a) Deleting a deleted pointer is undefined behavior, while deleting a nullptr does not do anything
   
   (b) A nullptr will cast to true of false

3. Use the `new` and `delete` operators to create allocate and delete heap memory for single objects and use `new[]` and `delete[]` to allocate and delete heap memory for arrays of objects.

4. The `new` operator can take in constructor parameters to create an object on the heap, whereas `new` can only default construct objects

As an example of the last point,

```cpp
int main() {
  int* a = new int; // create an int with random nonsense as value
  int* b = new int(); // create an int with 0 as value
  int* c = new int(1); // create a new integer with value 1
  std::string* d = new std::string; // create a string with "" as value
  std::string* e = new std::string("Hello world!"); // create a string with "Hello world!" as value
  int* z = new int[10](); // create a dynamic array of int with 0 as value
  // int* y = new int[10](1); // error! cannot initialize to a value using new[]
  std::string* x = new std::string[15]; // create a dynamic array of strings with value ""
```
2.2 Static vs dynamic arrays

A static array is allocated at compile-time, meaning that its relative address is memory is decided when the program is compiled (the compiler can do this because it knows how much space the array will take up for the duration of the program). Static arrays cannot be deleted and their size must be a constant, since the size must be known by the compiler. The following code creates a static array,

```cpp
constexpr unsigned sz = 100; // the size needs to be a constexpr so the compiler knows it will not change.
double array[sz];
```

A dynamic array is allocated at run-time, meaning that its relative address is memory is only decided when the program gets to the line of code declaring the array while running. Because the compiler doesn't need to know the size, dynamic arrays can be deleted and their size can be decided while the program is running. Dynamic arrays are stored on the heap. The following code creates a dynamic array

```cpp
#include <stdlib>  // srand, rand
#include <time>   // time

int main() {
    srand(time(nullptr)); // seed random number generator with current time
    double* arr = new double[rand()]; // create an array of random positive integer length
}
```

The array in the above code cannot possibly be compiler-allocated because its size depends on the current time, which is only known when the program is running.