Mandatory tasks

1. Create a new type called`Student` to represent students. Include ID, average, and age. The type of average should be a double-precision floating-point number, the rest integers. Examine the memory requirements of the type, experiment with the order of the data members and observe its effect.
2. For convenience, create an alias type with typedef so that you can use the type without the struct prefix.
3. Create a function that gets an `Student` array and returns the ID of the highest average student.
4. Treat BSc, MSc and PhD students separately, create an enum as Type and add it to the Student as a data member.
5. Depending on the type of student, store different extra data using a union type. BSc: total number of courses taken (int). MSc: Aggregate Adjusted Credit Index (double) PhD: the impact factor of a journal with the highest impact factor into which it has published (double) or Erdős number (int) using a struct. Let's look at the memory requirements of the type and compare what it would have been like to use struct instead.
6. Write a function (student_init) which, receiving a Type parameter, creates a`Student` instance on the heap with the appropriate Type, populates the corresponding data members with random data, and then returns a pointer to the instance.
7. Load an array with pointers to such `Student` instances, and then modify the function written in Task 3 to return a pointer to the instance instead of an identifier. Beware of memory leaks.

Optional tasks

1. For the matrix multiplication written in the previous lesson, create a `Matrix` type that contains the size of the dimensions of the matrix and a pointer to the first element of the array as a data member.
2. To facilitate indexing, write a `at` function that waits for a pointer, row and column index on a matrix instance, based on which it returns to the element marked by the indexes with a pointer.
3. Transform the matrix multiplication using these.

Advanced tasks

1. Create a dynamic array type with integers that stores the current size, capacity, and pointer to an array as a data member.

2. Create a function that creates a new instance based on the initial capacity received as a parameter, with properly configured data members. Then a pair of this, which appropriately releases the resulting copy.

3. Write a function that can be used to add an element to the end of an array. When the array is full, allocate an array twice as large, then copy the previous items and update the data members. Beware of memory leaks.