Multidimensional arrays in C

malloc

`malloc` is the standard memory allocation function in C. It returns a pointer to the beginning of a memory segment. Often `malloc` is used like this:

```c
int n_elements = 10;
int *arr = (int*) malloc(n_elements * sizeof(int));
```

This is fine, but there are some elements here we can change (for the better). It is unnecessary to cast the return value of `malloc`, so we can just remove the `(int*)` part. We can also use the `sizeof` operator a bit smarter. If we dereference the pointer `arr` using `*`, `sizeof` will be compiled to the size in memory of the type that `arr` points to. In this case a regular `int`.

```c
int n_elements = 10;
int *arr = malloc(n_elements * sizeof *arr);
```

This is a bit simpler. and if you need to change the type, eg. from `int` to `double`, you only need to perform the change at the start of the line.

And after you are done with an array remember to free the memory.

```c
free(arr);
```

For each call to `malloc` you should have a corresponding `free`.

2D arrays

In the memory of a computer there is no such thing as a multidimensional structure. All addresses in memory is essentially sequentially and 1D. If we want to represent a 2D structure we need to employ some tricks.

Pointer to pointer

A popular way to achieve syntax that resemble matrix notation is with pointers to pointers.

```
double **arr2d;
arr2d = malloc(m * sizeof *arr2d);
arr2d[0] = malloc(m*n * sizeof *arr2d[0]);

for (int i = 1; i < m; i ++) {
    arr2d[i] = &(arr2d[0][i*n]);
}
```

`arr2d` can now be used with the syntax `arr[i][j]`.

After we are done we need to free the memory

```
free(arr2d[0]); // Must be deallocated first to avoid memory leaks.
free(arr2d);
```

3D array

Allocation of the pointer to pointer to ... type arrays gets quite complicated. I prefer to not use this construction at all but rather index using some arithmetics. However, this construction is quite often used in code you will come across, so it's still useful to know what is going on.

```
int ***A;
A = malloc(nz * sizeof *A);
A[0] = malloc(nz*ny * sizeof *A[0]);

for (int i = 1; i < nz; i++) {
    A[i] = &A[0][i*ny];
}

A[0][0] = malloc(nz*ny*nx * sizeof *A[0][0]); // Contiguous

for (int j = 1; j < nz*ny; j++) {
    A[0][j] = &A[0][0][j*nx];
}
```

The first couple of lines, including the first for loop is exactly like for 2D arrays. Only the 2D array we get is type `int***` not `int**`. In the last part we add the final dimension. It's possible to allocate all the arrays first and "fix" the addresses later:

```c
int ***A;
A = malloc(nz * sizeof *A);
A[0] = malloc(nz*ny * sizeof *A[0]);
A[0][0] = malloc(nz*ny*nx * sizeof *A[0][0]); // Contiguous

for (int i = 1; i < nz; i++) {
    A[i] = &A[0][i*ny];
}

for (int j = 1; j < nz*ny; j++) {
    A[0][j] = &A[0][0][j*nx];
}


// Assign some values:
for (int i = 0; i < nz; i++) {
    for (int j = 0; j < ny; j++) {
        for (int k = 0; k < nx; k++) {
            A[i][j][k] = nx * ny * i + ny * j + k;
        }
    }
}
```

And again, remember to free the memory

```c
free(A[0][0]); // Free the main contiguous block first.
free(A[0]);
free(A);
```

Alternative

We can avoid all the complicated pointer to pointer arrays and instead opt for a simpler solution. The underlying memory don't care if we index using pointers or directly into to the final 1D contiguous block.

```
#define idx(i,j) (i*ny + j)

int *arr = malloc(nx*ny * sizeof *arr);

for (size_t i = 0; i < nx; i++) {
  for (size_t j = 0; j < ny; j++) {
    arr[idx(i,j)] = i + j;
  }
}

free(arr);
```

And for 3D

```
#define idx(i,j,k) (i*ny*nz + j*nz + k)

int *arr = malloc(nx*ny*nz * sizeof *arr);

for (size_t i = 0; i < nx; i++) {
  for (size_t j = 0; j < ny; j++) {
    for (size_t k = 0; k < nz; k++) {
      arr[idx(i,j,k)] = i + j + k;
    }
  }
}

free(arr);
```

This syntax is much easier to use and debug (in my opinion). We also avoid allocating space for the pointers. Note that `ny` and `nz` must exist in the context you use the defined function. Here i use them in the loops as well.

valgrind

Now that we are using arrays with multiple dimensions it is useful to check our program for memory leaks. Valgrind is a program we can use for this purpose.

```
$ valgrind --leak-check=full ./myprog arg1 arg2 ...
```

Will run a check for memory leaks. Search google for a complete tutorial if you are interested, but this simple line is often all you need.

Define

Defines does not act like regular functions in C. Defines are inserted into your code by the preprocessor before the program is compiled. Code that looks like this:

```c
#define idx(i, j) i*ny + j

arr[idx(i, j)] = 5;
```

Will after the preprocessor is done with it read

```c
arr[i*ny + j] = 5;
```

You therefore need to be careful when defining things other than constants. The result might not be what you intend. Ex.:

```c
#define ADD(a, b) a + b

double  term = ADD(2, 5)/3;
```

Gets converted into:

```c
double  term = 2 + 5/3;
```

The order of operations is not what was intended. This issue is solved by using parenthesis.

```c
#define ADD(a, b) (a + b)
```