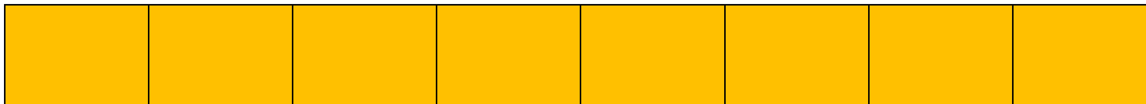
FAQ Ordered ArrayList in C

Q1 What's really going on here?

A Physically, the `arrayList` is just a large block of N bytes of memory:

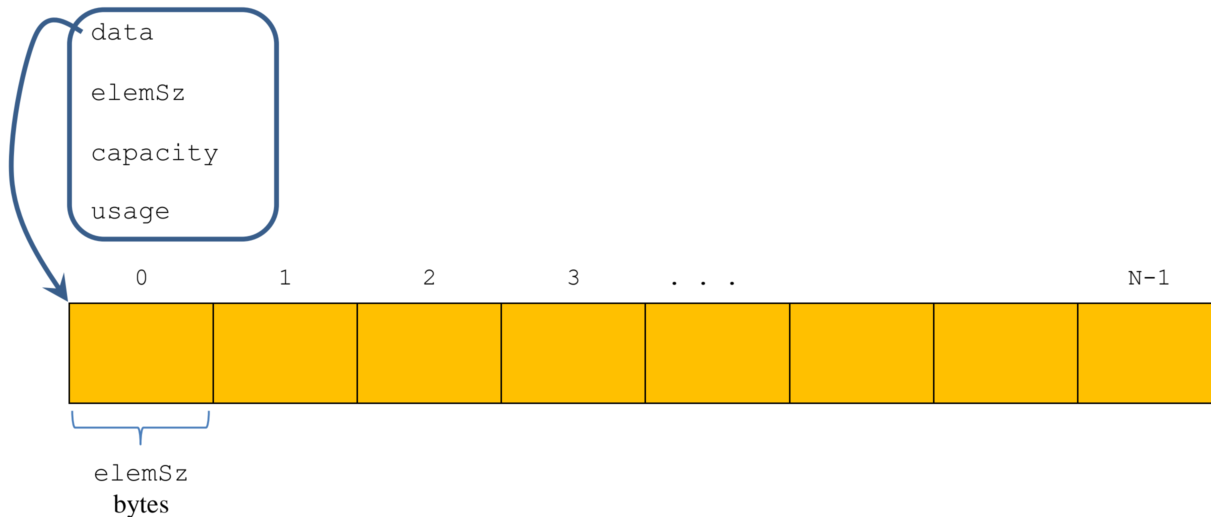
Logically, we think of that block of memory as being divided into sequential "cells" of the same size:



Each cell can hold one user data object, but the user has to tell us how big one of those data objects is, and how many data objects the user wants to put in, so that:

- We know how big to make the whole memory block.
- We know how many bytes we need to copy when we put a user data object into a cell.
- We know how to move from one cell to the next, or how to compute the address where a given cell begins.

Then an `arrayList` object looks something like this:



Each cell can store whatever kind of user data object is being supplied, as long as the user tells us the correct size for those elements. For flexibility, we'll have a default size for the `arrayList` memory block, and switch to a larger memory block as needed, as the user inserts data objects.

Q2 How do we "find" a specific cell if we are given its index?

A Short answer: *pointer arithmetic*. So see the discussions of that in the course notes, and in the Appendix on array logic. All the basic facts you need are given there. Really… they are.

One suggestion: I wrote a helper function for my `arrayList` that takes a cell index and returns the address where that cell starts in the memory block. That helps to isolate the pointer arithmetic to a single spot in my code, and makes it easier to think about the manipulations in terms of classic array indexing (more or less).

Q3 So, given a pointer to a user data object and a cell index, how do I put the user's data object into the cell?

A Use `memcpy()`, which is described in the specification.

Q3b Why can't I just use the assignment operator to copy the user data object, something like this (syntax may be imprecise):

```
pAL->data[index] = *ptrToUserDataObject;
```

A The operation of the dereference operator (`*`) depends on the size of the target of the pointer. Dereferencing a pointer "yields" a chunk of bytes equal in size to the type of the pointer's target. Dereferencing a `uint32_t*` yields 4 bytes. Dereferencing a `MLBPerson*` yields a number of bytes equal to the size of an `MLBPerson` object, and that works fine since if we have a `MLBPerson*` and the compiler has seen the declaration of the `MLBPerson` type.

But, with a generic implementation of `arrayList`, we don't know anything about the user's data type. All we have is a `void*`, and the specification discusses the fact that `sizeof(void*)` is ill-defined, and would not be likely to give us a useful value anyway.

Hence, we use `memcpy()`.

Q4 Why does the `arrayList` store user data objects physically, instead of storing pointers to the user's data objects?

A Because:

- That involves an extra pointer dereference to access the user's data objects.
- That requires that the user not do anything disruptive, like deallocate data objects while those objects are "in" the arrayList.

The `ArrayList` in Java does store pointers to user data objects, but then Java only allows us to access objects of a class via a reference.

Q5 If I have a `uint8_t*`, how can I add a value bigger than a `uint8_t` to it without triggering an overflow (and disastrous results)?

A A `uint8_t*` is represented as an 64-bit value, which happens to be the address of a `uint8_t`. Don't confuse the pointer with its target. There's no concern about creating an overflow.

As for storing a value that's wider than a `uint8_t` at the target of a `uint8_t*`, that's a matter of using pointer typecasts, which are discussed in the course notes for the week of Sept 27. The following statements will copy one byte of data:

```
uint8_t* p = address of some target;
uint8_t* q = address of some target;
*p = *q;
```

On the other hand, these statements will copy 8 bytes of data:

```
uint8_t* p = address of some target;
uint8_t* q = address of some target;
*p = *(uint64_t*)q;
```

Of course, you'd better make sure that there is enough room, starting where `p` points, to hold 8 bytes.

But, having said all this, see the note about using `memcpy()`. Typecasting requires knowing the type we want to cast to, and with the `arrayList`, we never know the actual type of the user's data object.