

# CS 2213, Spring 2009

## Dynamic Array Lists

These notes are meant to supplement the lecture in CS 2213 on 3/6/09 and 3/16/09. They discuss an implementation of “Dynamic Array Lists” (a.k.a. dynamic lists, dynamic arrays, growable arrays, resizable arrays, dynamic tables, and array lists) in C99. They are meant to be used along with the `dynamic-reverse` program example as an aid in developing a solution to Assignment 5.

### Introduction

Normally, arrays must either be created with a fixed size (either the size they are declared with at compile-time<sup>1</sup> or the size passed to `malloc` when they are dynamically allocated). But sometimes one needs to be able to store an arbitrary number of items without knowing in advance without know how many items there should be.

One data structure that is capable of handling this is the *dynamic array list*.<sup>2</sup> The main idea is to enclose a primitive array into a larger structure, such that the array can be replaced with a larger one when the amount of data grows too large for the current array.

The most straight forward implementation is to have a dynamically allocated array and then bundle up the variables needed to manage that array into a structure declared something like.

```
struct darray1 {
    int size;        // the number of elements currently stored in data
    int capacity;   // the number of elements for which there is space in data
    char *data;     // a pointer to the start of a dynamically allocated array
}
```

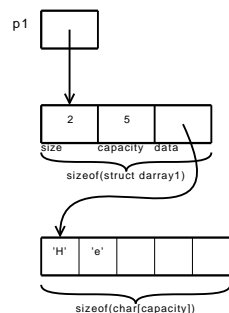
This could then be initialized to create an empty dynamic array list with initial capacity of 5 with something like:<sup>3</sup>

```
struct darray1 *p1 = malloc(sizeof(struct darray1));
p1->size = 0;
p1->capacity = 5; // initial capacity
p1->data = malloc(p1->capacity * sizeof(char));
```

One can then append the elements 'H', 'e', with something like:

```
p1->data[p1->size++] = 'H';
p1->data[p1->size++] = 'e';
```

At this point the resulting memory would look like:



1. This is relaxed for C99 variable length arrays, which allow variable expressions to occur in our array declaration, making the length of automatic array variables as flexible as `malloc`'ed variables.
2. Version of the dynamic array data structure can be found among Java's container classes (the `java.util.ArrayList` class) and in C++'s standard template library (the `vector` class).
3. I'm omitting error-checking and any kind of more general usability. The examples directory has full worked out example for the `darray2` variant and its use at character handling.

One could, furthermore, add 'l', 'L', 'o',! to it using

```
p1->data[p1->size++] = 'l'
p1->data[p1->size++] = 'L'
p1->data[p1->size++] = 'o'
```

At this point, the size would be 5, the same as the capacity. To add a sixth character, e.g., '!'. It would be necessary to first obtain a larger block of dynamically allocated memory and copy the contents of the current data over to it. Omitting, error checking, this could be done with something like:

```
if (p1->size == p1->capacity) {
    char *old_data = p1->data;
    p1->capacity *= 2;
    p1->data = malloc(p1->capacity * sizeof(char));
    for (int i = 0; i < p1->size; i++)
        p1->data[i] = old_data[i];
    free(old_data);
}

p1->data[i] = '!';
```

## Relevant C Language/Library Features

There are a couple of C language/library features that can be applied to improve upon this idea.

*malloc()*. First of all, in addition to *malloc()* the C library provides a function *realloc()* which can be used to replace a dynamically allocated chunk of memory with an larger or smaller one with the same contents. (If possible it just changes the size of the existing memory chunk, but if this isn't possible, it will allocate new space, copy the array to the new space, and free up the old space. Thus the code:

```
p1->data = malloc(p1->capacity * sizeof(char));
for (int i = 0; i < p1->size; i++)
    p1->data[i] = old_data[i];
free(old_data);
```

Could have been replaced with:

```
p1->data = realloc(p1->data, p1->capacity * sizeof(char));
```

(Note that, like *malloc()*, *realloc()* can fail and return NULL. In this case memory pointed to by *realloc()*'s first argument will remain unmodified. See the full example in subversion for a worked out example of how to handle such errors.)

*C99 Flexible Array Members*. For some uses, the code for `struct darray1` above may be considered inelegant since it splits the data for the dynamic array list across to different chunks of memory (one for the structure and another for the array). C99 provides a feature that can be applied to the dynamic array list is the C99 Flexible Array Member to put everything into a single chunk of memory.

In C99, one is allowed to declare a lengthless array (the "Flexible Array Member") as the last field (let's call this element `data`) of a structure. If this is done, then as long as the structure is created using a *malloc()* or *realloc()* large enough to store the full structure containing `data`, plus `n` element array of `data`'s element type, then C99 allows one to use the `data` field of the structure as an `n` element array. We could modify our dynamic array list structure declaration to use flexible array members as follows:

```

struct darray2 {
    int size;
    int capacity;
    char data[];
}

```

We could then initialize a darray2 dynamic array list with an initial capacity of 5, as follows:<sup>4</sup>

```

struct darray2 *p2 = malloc(sizeof(struct darray2) + 5*sizeof(char));
p2->size = 0;
p2->capacity = 5;

```

This has the advantage that all of the pieces of the dynamic array (its size, its capacity, and its data elements) are all in one chunk of memory.

The code to add an element to p2, would look something like:<sup>5</sup>

```

if (p2->size == p2->capacity) {
    p2->capacity *= 2;
    p2 = realloc(p2, sizeof(struct darray2) + p2->capacity * sizeof(char));
}
p2->data[p2->size++] = 'H';

```

Note that, now, `realloc()` may need to move the entire structure in order to increase the size of `data[]`. This requires the ability to assign to `p2`. This can be a problem if, this code were to be encapsulated into a function and `p2` were to be passed as a parameter, since the caller's copy of the `p2` would not be updated by the assignment to `p2`. One solution for this is for the function to pass a pointer to the pointer to `p2`, and use that to update `p2`.<sup>6</sup>

```

void da2_add(struct darray2 **pp, char element) {
    if (*pp->size == *pp->capacity) {
        int nsize;

        *pp->capacity *= 2;
        nsize = sizeof(struct darray2) + *pp->capacity * sizeof(char);
        *pp = realloc(*pp, nsize);
    }
    *pp->data[*pp->size++] = element;
}

```

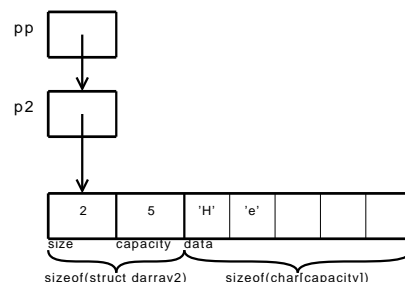
This could be called with:

```

da2_add(&p2, 'e');

```

After the execution of the code above, the memory associated with the darray2 structure would look like:



4. Again, omitting the necessary error handling.  
5. Again, omitting error handling code.  
6. Again, omitting error handling code.

Flexible Array Members is a new feature in C99. There is, however, a similar ‘trick,’ using `data[1]` instead of `data[]`, that, though violating standard defined pointer arithmetic, works on most C89 and C99 compilers.

## Interface

In a large software system, it is useful for the client (i.e., the rest of the program except for the implementation of the dynamic array) to be able to treat the entire dynamic array as a single entity that appears to automatically grows as necessary.

The operations, which can be accomplished efficiently with a dynamic array list include:

- create a new (empty) dynamic array list
- append a new element to the end of the dynamic array list
- get the  $n^{\text{th}}$  element
- remove the last element
- delete/free the memory for a dynamic array list

An interface for these operations can be put into a header file containing the following declarations:

```
struct darray2;

struct darray2 *da2_init(int capacity);           // allocates/initializes darray2
int da2_add(struct darray2 **dca, char item);    // appends a character
int da2_size(struct darray2 *dca);              // returns the number of elements
char *da2_get(struct darray2 *dca, int index);  // gets the index'th element
void da2_remove(struct darray2 *dca);           // removes the last element
void da2_free(struct darray2 *dca);             // frees memory for a darray2
```

The usage of these functions, should be mostly self-explanatory. The `da2_add` function requires a pointer-to-pointer to a `darray2` structure because it needs to be able to modify the caller’s pointer to a `darray2` if it moves the `darray2` structure as part of a `realloc()` as described above. The `da2_get()` function provides random access to an arbitrary element of the dynamic array list. It returns a pointer to the element rather than the element itself, since the dereference pointer can be used to both read the element and to modify an element; thus it isn’t necessary to have a separate set function.

## Implementation Invariants

As described above, the dynamic list can be represented using three fields bundled into a C structure:

- the number of elements currently stored in the dynamic array (*size*).
- the capacity of the primitive array (*capacity*), and
- the actual dynamically sized primitive array (*data*),

The important invariants of these fields are that :

- $size \leq capacity$  and
- the dynamically allocated memory is large enough that *data* can hold *capacity* elements of *data*.

## Dynamic Reverse Example Implementation

A full example dynamic character array list implementation providing this interface (with slightly different function names) can be found as part of the `dynamic-reverse` program contained in the `examples/dynamic-reverse` directory. This is a program that reads in an arbitrarily long string of characters (as long as the string fits into memory, and its length is less than half `INT_MAX`) and prints them out in reverse order. As each character is read in, this program places the character in a dynamic array list of characters.

Unlike some other programming languages, C does not provide a good mechanism for designing generic data structures that can be instantiated for specific data types as needed (cf. generics in Java, templates in C++, polymorphic types and functors in ML). Therefore, in order to have a list of a data type other than characters (as you will need to do for assignment 5), you will need to change all references to `char` to whatever the type of the element type of the dynamic array list should be. (You will probably also want to change the names, since the `c`'s stand for `char`.)

*Enlargement Factor* There is one last item, these notes will discuss. How much should the dynamic array be enlarged by each time its capacity is exceeded? In the code above, and in the code in the `dynamic-reverse` example, we have doubled the capacity each time, more capacity is needed. This is not strictly necessary. Everything will work and no space will be wasted if one just increases the capacity by one each time the capacity is exceeded, but that would make the *append new element* operation very expensive, since the whole array might need to be copied each time an element is added. A better option, at least as far as asymptotic performance is concerned, is to sacrifice some amount of space proportion to the amount of data by multiplying the capacity by some constant *enlargement factor* (such as the 2 used here) each time; this way, as the cost of copying the entire primitive array gets more expensive, it happens less frequently. The precise "best" value for this *enlargement factor* and for the *initial capacity* of each dynamic array will depend on how the dynamic array will be used. In any case, an *enlargement factor* of two makes a reasonable default.