Worksheet 14: Introduction to the Dynamic Array

In Preparation: Read Chapter 5 to learn more about abstraction and the basic abstract data types, and the dynamic array as a n implementation technique.

Suppose you want to write a program that will read a list of numbers from a user, place them into an array and then compute their average. How large should you make the array? It must be large enough to hold all the values, but how many numbers will the user enter? A common solution to this difficulty is to create an array that is larger than necessary, and then only use the initial portion. This is termed a *partially filled array*.

When you use a partially filled array there are two integer values of interest. First is the array length. When discussing partially filled array this is sometimes termed the *capacity* of the array. Second is the current *size*, that is, the amount of the array that is currently being used. The size is generally maintained by a separate integer variable. If we place the values into a structure it will be easier to keep them together.

It is important to not confuse the size and the capacity. For example, in computing the sum and average you do not want to use the capacity:

```
double average (struct partFillArray * pdata) {
  double sum = 0.0;
  int i;
  for (i = 0; i < 50; i++)  /* Error–loop using length */
          sum = sum + pdata->data[i]; /* Error-uninitialized value */
  return sum / 50; /* Error–average is under estimated */
}
```

Instead, you want to use the size:

```
double average (struct partFillArray * pdata) {
  double sum = 0.0;
  int i;
  for (i = 0; i < pdata->size; i++)
          sum = sum + pdata->data[i];
  return sum / pdata->size;
}
```

The technique of partially filled arrays works fine until the first time that the user enters more numbers than were originally anticipated. When this happens, the size can exceed the capacity, and unless some remedial action is taken an array indexing error will occur. Worse yet, since the validity of index values is not checked in C, this error will not be reported and may not be noticed.

A common solution to this problem is to use a pointer to a ***dynamically allocated array***, rather than a fixed length array as shown above in the partFillArray. Of course, this means that the array must be allocated before it can be used. We can write an initialization routine for this purpose, and a matching routine to free the data. Next, we likewise encapsulate the action of adding a new element into a function. This function can check that the size does not exceed the capacity, and if it does increase the length of the array (generally by doubling) and copying all the elements into the new area. Now the user can enter any number of values and the data array will be automatically expanded as needed.

```
# ifndef TYPE
# define TYPE int
# endif

struct DynArr
{
	TYPE *data;	/* pointer to the data array */
	int size;		/* Number of elements in the array */
	int capacity;	/* capacity of the array */
};

void initDynArr(struct DynArr *v, int capacity)
{
	v->data = malloc(sizeof(TYPE) * capacity);
	assert(v->data != 0);

	v->size = 0;
	v->capacity = capacity;

}
void freeDynArr(struct DynArr *v)
{
	if(v->data != 0)
	{
		free(v->data); 	/* free the space on the heap */
		v->data = 0;   	/* make it point to null */
	}
	v->size = 0;
	v->capacity = 0;
}



int sizeDynArr( struct DynArr *v)
```

```
{
        return v->size;
}

void   addDynArr(struct DynArr *v, TYPE val)
{
        /* Check to see if a resize is necessary */
        if(v->size >= v->capacity)
                _setCapacityDynArr(v, 2 * v->capacity);

        v->data[v->size] = val;
        v->size++;
}
```

The only thing missing now is the _setCapacityDynArr function. Complete the implementation of _setCapacityDynArr. Pay careful attention to the order of operations. Remember that since you're creating a new array, you'll want to eventually get rid of the old one to avoid a 'memory leak'.

```
 void _setCapacityDynArr(DynArr *v, int newCap)
{


        TYPE *newData;  /* pointer to new array */
        int oldCap;     /* current capacity of v's array */
        int i;

        assert(v != NULL);
        oldCap = v->capacity;
        assert(newCap >= oldCap);

        /* allocate new array on heap */
        newData = (TYPE *)malloc(sizeof(TYPE) * newCap);
        assert(newData != NULL);

         /* copy elements from old array to new array */
        for (i = 0; i < oldCap; ++i)
                newData[i] = v->data[i];

        free(v->data);      /* deallocate old array */

        /* update v's members to use the new array and capacity */
        v->data = newData;
        v->capacity = newCap;
}
```

The code shown above introduces a number of features of the C language that you may not have seen previously. Let us describe some of these here.

TYPE. We want to create a library of general-purpose functions for managing collections of various types of elements. In order to make our code completely independent from the type of value being stored, we have defined the element type using a symbolic name, TYPE. The use of ifdef surrounding the definition is a common C idiom. If the user has already provided an alternative definition we will use that, otherwise the symbolic name is given a default value of int.

initDynArr and freeDynArr. The C language does not provide a way to automatically initialize a structure, such as a constructor does for you in Java or C++. Instead, programmers must typically write a special initialization function. Programmers must then remember to initialize a structure, and to free memory when they are finished with the structure.

```
struct DynArr myData;  /* create a new dynamic array */
…
initDynArr (&myData, 50); /* initial capacity is 50 elements */
…
freeDynArr (&myData);
```

Notice that the structure is declared as a simple variable. However, because the functions require a pointer, the *address-of* operator (the ampersand) is used to produce a pointer to the structure.

da->size. Whenever you use a pointer you must make it clear when you are referring to the pointer itself and when you are referring to the value it points to. Normally a pointer value must first be dereferenced, using the * operator, to indicate that you mean the value it points to, not the pointer itself. Accessing a field in a structure referred to by a pointer could be written using the dereference operator, as in (*da).size. However, this combination of pointer dereferencing and field access occurs so frequently that the designers of C provided a convenient shorthand.

malloc. The function malloc is used to perform *m*emory *alloc*ation in C. The argument is an integer indicating the number of bytes requested. In order to determine how many bytes are required for each element the function sizeof is invoked. Multiply the number of elements you need by the size of each element, and you have a block of memory that can be used as an array.

assert. The malloc function will return zero if there is insufficient memory to satisfy a request. The assert macro will halt execution with an error message if its argument expression is not true. Assertions can be used any time a condition must be satisfied in order to continue.

free. The function free is the opposite of malloc. It is used to return a block of memory to the free store. Such memory might later be reused to satisfy a subsequent malloc request. You should never use malloc without knowing where and when the memory will subsequently be freed.

Defensive programming. When the memory is released in the function freeDynArr the size and the capacity are both set to zero. This ensures that if a subsequent attempt is made to insert a value into the container, there will not be an attempt to index into an already deleted array.

sizeDynArr. Since the size field is stored as part of the dynamic array structure there really is no need for this function, since the user can always access the field directly. However, this function helps preserve encapsulation. The end user for our container need not understand the structure definition, only the functions needed to manipulate the collection. *How would your code have to change in order to completely hide the structure away from the user?*

_setCapacityDynArr. An underscore is treated as a legal letter in the C language definition for the purposes of forming identifiers. There is a common convention in the C programming community that function names beginning with an underscore are used “internally”, and should never be directly invoked by the end user. We will follow that convention in our code. The function _setCapacityDynArr can be called by dynamic array functions, but not elsewhere.

Note carefully the order of operations in the function _setCapacityDynArr. First, the new array is created. Next, the old values are copied into the new array. The free statement then released the old memory. Finally, the pointer is changed to reference the new array.

In order to allow a dynamically allocated array to be used in the same fashion as a normal array, we need functions that will get and set values at a given position. We can also make our function more robust than the regular C array by checking index positions. Complete the implementation of the following functions. *Use assert to check that index positions are legal*.

```
TYPE getDynArr (struct DynArr * da, int position) {

        assert(da != NULL);
        assert(da->size > 0);
        assert(position >= 0);
        assert(position < da->size);

        return da->data[position];

}

void putDynArr(struct DynArr * da, int position, TYPE value) {

        assert(da != NULL);
        assert(da->size > 0);
        assert(position >= 0);
        assert(position < da->size);

        da->data[position] = val;

}
```

Write the function swap, which will exchange the values in two positions of a dynamic array. We will use this function in later chapters.

```
void swapDynArr (struct DynArr * da, int i, int j) {

        TYPE temp = da->data[i];
        da->data[i] = da->data[j];
        da->data[j] = temp;

}
```

Write the function removeAtDynArr, which will remove a value at a specified index. Remember, we do not want to leave gaps in the partially filed array. We will use this function in later chapters.

```
void removeAtDynArr (struct DynArr * da, int index) {
        for (int i = index; i < da->size - 1; i++) {
                da->data[i] = da->data[i + 1];
        }
        da->size--;

}
```