

Dynamic Arrays Part 2: Implementation

Arrays: Pros and Cons

- Pro: only core data structure designed to hold a collection of elements
- Pro: random access: can quickly get to any element $\rightarrow O(1)$
- Con: fixed size:
 - Maximum number of elements must be specified when created

Element Types

- How to make a general purpose container class?
- We define TYPE as symbolic preprocessor constant
- Requires recompiling source for new element types
 - Not elegant, but workable.

Interface File: `dynarr.h`

```
#ifndef __DYNARR_H
#define __DYNARR_H

# define TYPE int

# define LT(a, b) ((a) < (b))

# define EQ(a, b) ((a) == (b))

... /* Rest of dynarr.h (on next slide). */
#endif
```

Interface (cont.)

```
struct DynArr {  
    TYPE *data; /* Pointer to data array. */  
    int size; /* Number of elements in collection. */  
    int cap; /* Capacity of array. */  
};  
  
/* Dynamic Array Functions */  
void initDynArr(struct DynArr *v, int cap);  
void freeDynArr(struct DynArr *v);  
int sizeDynArr(struct DynArr *v);  
void addDynArr(struct DynArr *v, TYPE e)  
TYPE getDynArr(struct DynArr *v, int pos);  
void putDynArr(struct DynArr *v, int pos, TYPE  
    val);
```

Initialization: `initDynArr`

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

Clean Up: freeDynArr

```
void freeDynArr(struct DynArr *v) {  
    if(v->data != 0)  
    {  
        free(v->data) ;  
        v->data = 0;  
    }  
    v->cap = 0;  
    v->size = 0;  
}
```

Concretely, in C...using the dynArray

```
struct DynArr d;  
initDynArr(&d, 8);  
addDynArr(&d, 1);  
...
```


Better Solution

To use a struct dynArr, the user must declare one (see previous slide). To declare it, the compiler must know its size when compiling that file (ie. it must be in the header!)

If it's in the header, it is 'exposed' to the end user and **this can be dangerous and violates 'encapsulation'**

Better Solution: Provide create() and delete() functions for your data structure. New returns a 'pointer' to allocated space

User can always declare pointers and compiler always knows the size of a pointer! Now you can hide your Struct in the .c file or a library

Create/Delete

```
DynArr* createDynArr(int cap)
```

```
{
```

```
    DynArr *r;
```

```
    assert(cap > 0);
```

```
    r = malloc(sizeof( struct DynArr));
```

```
    assert(r != 0);
```

```
    _initDynArr(r, cap);
```

```
    return r;
```

```
}
```

Allocate space for struct
DynArr itself!



When to use Dynamic Arrays

- Need random access
- Low memory footprint
- Don't know size of array at compile time
- See Examples in Java and C++ STL
 - Vector (C++ STL)
 - Vector and ArrayList (Java)
- When should/should not use a dynamic array!
 - When $O(N)$ resize is NEVER acceptable

Dynamic Array Stack/Bag

- First: Worksheet 14 – Dynamic Array Basics
 - `_setCapacity`
 - Get, Put
 - Swap , RemoveAt
- Worksheets 16, 21
 - Implement the stack/bag interfaces
 - keep and reuse functionality from WS#14 where appropriate.