

resizable array

Resizable Array

- Think about a set of functions that provide a mechanism of resizable array of int.
 - Growable
 - Get the current size
 - Access to the elements

the Interface

- `Array array_create(int init_size);`
- `void array_free(Array *a);`
- `int array_size(const Array *a);`
- `int* array_at(Array *a, int index);`
- `void array_inflate(Array *a, int more_size);`

the Array

```
typedef struct {  
    int *array;  
    int size;  
} Array;
```

Why struct not struct *?

array_create()

```
Array array_create(int init_size) {  
    Array a;  
    a.array = (int*)malloc(sizeof(int)*init_size);  
    a.size = init_size;  
    return a;  
}
```

Why Array not Array *?

array_free()

```
void array_free(Array *a) {  
    free(a->array);  
    a->size = 0;  
}
```

array_size()

```
int array_size(const Array *a) {  
    return a->size;  
}
```

Why not take the member directly?

array_at()

```
int* array_at(Array *a, int index) {  
    if ( index >= a->size ) {  
        array_inflate(a, index-a->size);  
    }  
    return &(a->array[index]);  
}
```

Why int* not int?

use array_at()

```
Array a = array_create(10);
```

```
*(array_at(&a, 5)) = 6;
```

```
*(array_at(&a, 10)) = *(array_at(&a, 5));
```

will it be better

- to have two access functions:
 - `array_get()`, and
 - `array_set()`

use get() and set()

```
Array a = array_create(10);
```

```
array_set(&a, 5, 6);
```

```
array_set(&a, 10, array_get(&a, 5));
```

memory in block

```
int* array_at(Array *a, int index) {  
    if ( index >= a->size ) {  
        array_inflate(a, (index/BLOCK_SIZE  
+ 1)*BLOCK_SIZE-a->size);  
    }  
    return &(a->array[index]);  
}
```

array_inflate()

```
void array_inflate(Array *a, int more_size) {  
  
    int* p = (int*)malloc(sizeof(int)*(a->size  
+more_size));  
  
    for ( int i=0; i<a->size; i++ ) p[i] = a->array[i];  
  
    free(a->array);  
  
    a->array = p; a->size = a->size+more_size;  
  
}
```

array_inflate()

```
void array_inflate(Array *a, int more_size) {  
    int* p = (int*)malloc(sizeof(int)*(a->size  
+more_size));  
  
    memcpy((void*) p, (void*) a->array, a-  
>size*sizeof(int));  
  
    free(a->array);  
  
    a->array = p; a->size = a->size+more_size;  
  
}
```

why not take the whole array

```
int* array_get(Array* a) {  
    return a->array;  
}
```

lack of protection for both
user and developer

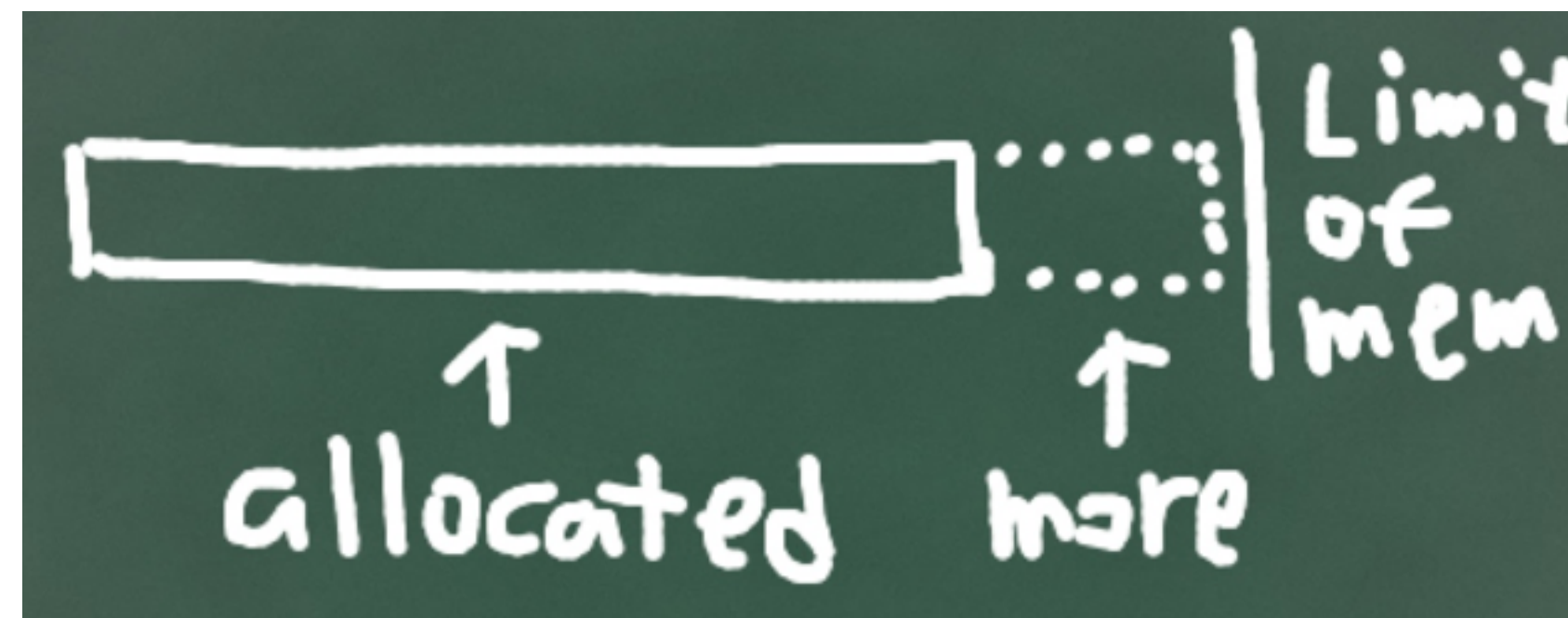
access functions

- the use of access functions seems not so elegant
- Use operator overload in C++
- Design specific functions for specified application
 - Do not treat it as an array

linked-array

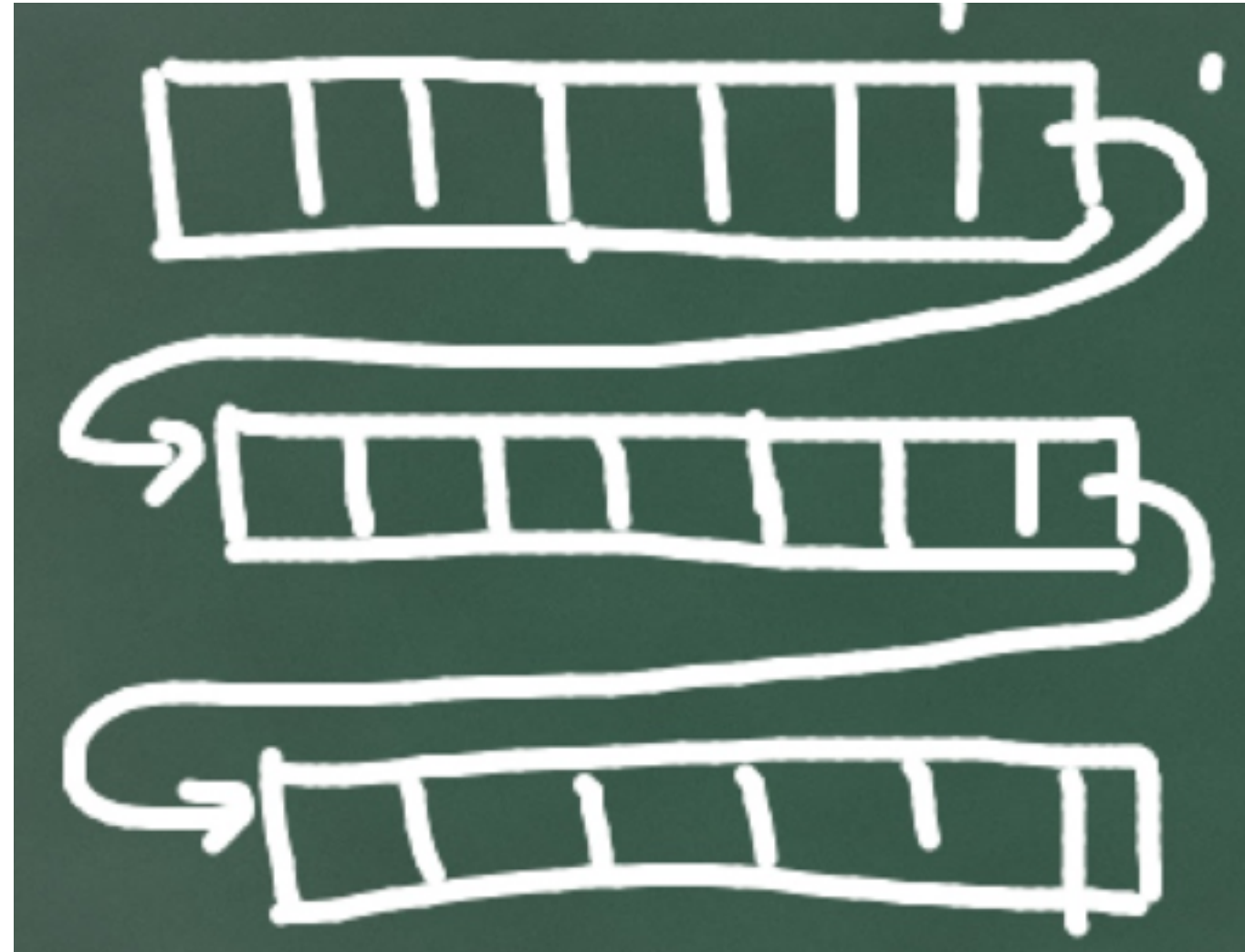
issues

- Allocate new memory each time it inflates is an easy and clean way. But
 - It takes time to copy, and
 - may fail in memory restricted situation



linked blocks

- No copy



the Array

```
typedef struct _array{  
    int *array;  
  
    int size;  
  
    struct _array* next;  
  
} Array;
```

array

use a fixed block
size, but keep the
variable to make it
more flexible

array_create()

```
Array array_create() {
```

```
    Array a;
```

```
    a.array = (int*)malloc(sizeof(int)*BLOCK_SIZE);
```

```
    a.size = BLOCK_SIZE;
```

```
    a.next = 0;
```

```
    return a;
```

```
}
```

array_free()

```
void array_free(Array *a) {  
    free(a->array);  
    a->size = 0;  
    if ( a->next ) {  
        array_free(a->next);  
        free(a->next);  
    }  
}
```

array_size()

```
int array_size(const Array *a) {  
    if ( !a->next )  
        return a->size;  
    else  
        return a->size+array_size(a->next);  
}
```

array_at()

```
int* array_at(Array *a, int index) {  
    if ( index < a->size ) {  
        return &(a->array[index]);  
    } else {  
        ...  
    }  
}
```


array_inflate()

```
void array_inflate(Array *a) {  
    // find the last block  
  
    // allocate a new block  
  
    // link!  
  
}
```