DYNAMIC ARRAYS

PARTIALLY FILLED ARRAY

Concept

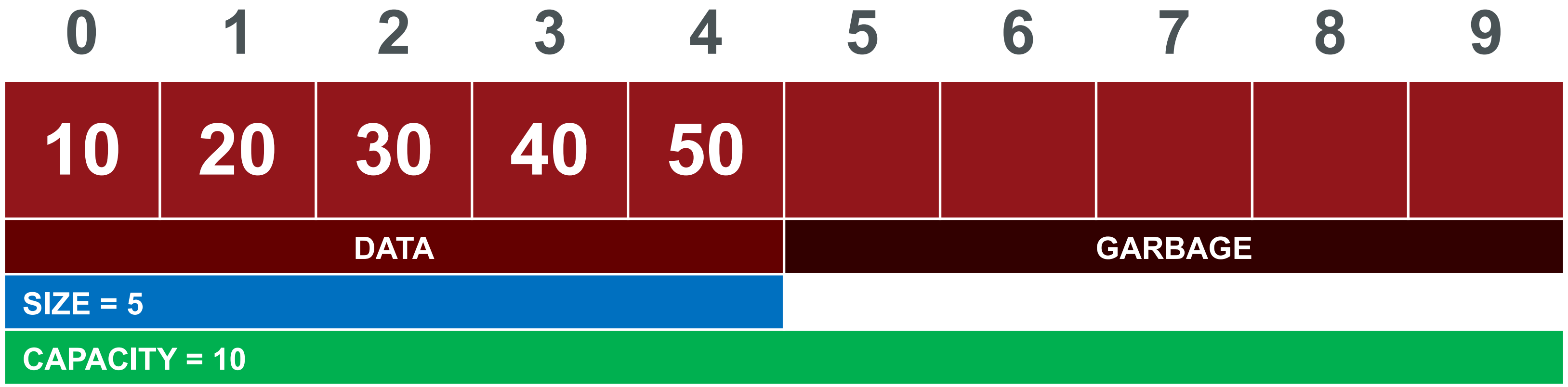
an array that is not filled with values
a partially filed array requires two variables size and capacity
array values must be in adjacent memory cells starting from index 0 (no holes)

Size — the current number of values in an array

Capacity — the maximum number of values in an array (allocated memory)

Example

```
const int CAPACITY = 10;                    // maximum number of values
int a[CAPACITY] = {10, 20, 30, 40, 50};     // partially filled array
int size = 5;                               // current number of values
```



0	1	2	3	4	5	6	7	8	9
10	20	30	40	50					

DATA

GARBAGE

SIZE = 5

CAPACITY = 10

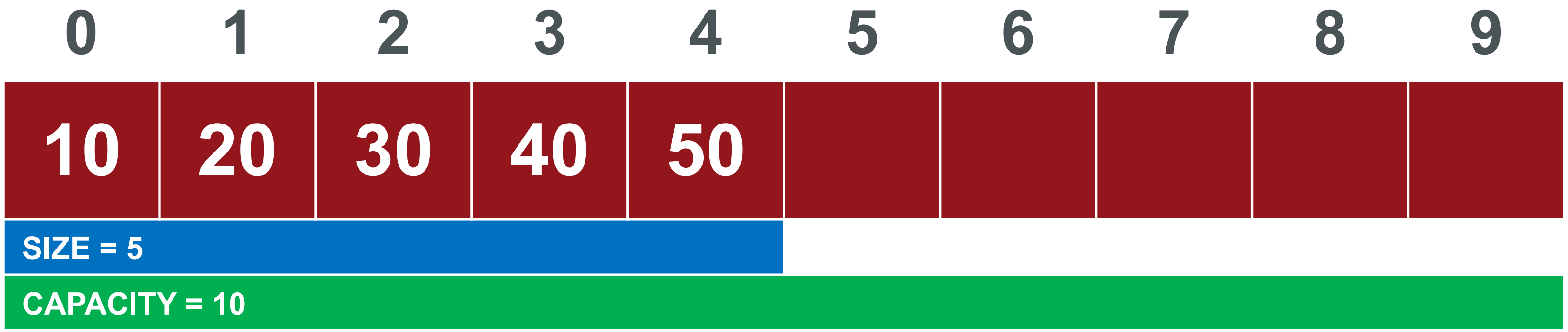
PARTIALLY FILLED ARRAY: PRINT

Print iterate until size not capacity, size represents the amount of data

Example

```
void print(int *a, int size) {
        for(int i=0; i<size; ++i) {                    // iterate through the array
                cout << a[i] << “ “;
        }
}

print(a, size);                                        // print size values
```



0	1	2	3	4	5	6	7	8	9
10	20	30	40	50					

SIZE = 5

CAPACITY = 10

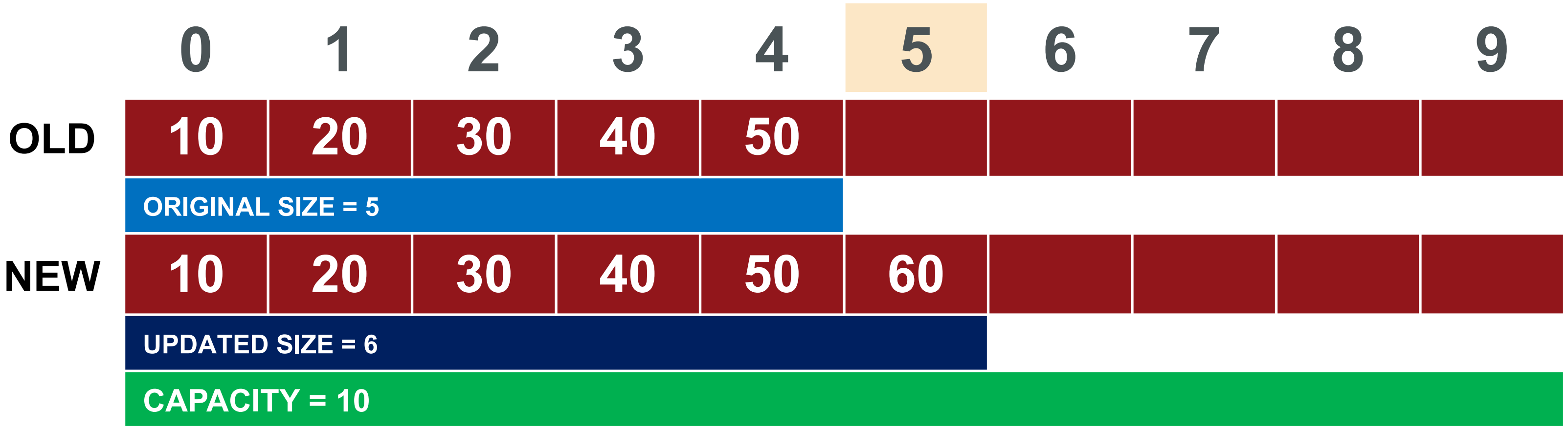
PARTIALLY FILLED ARRAY: PUSH_BACK

Concept store a value at the end of the array

Example

```
void push_back( int *a, int capacity, int &size, int value ) { // size pass by reference
    if( size < capacity ) {                        // if there is room in the array
        a[size] = value;                           //     store 60 at a[5]
        ++size;                                    //     increment size to 6, since 60 added
    }
}

push_back(a, CAPACITY, size, 60);                  // store 60 at the end of the array
```



	0	1	2	3	4	5	6	7	8	9
OLD	10	20	30	40	50					
NEW	10	20	30	40	50	60				

ORIGINAL SIZE = 5

UPDATED SIZE = 6

CAPACITY = 10

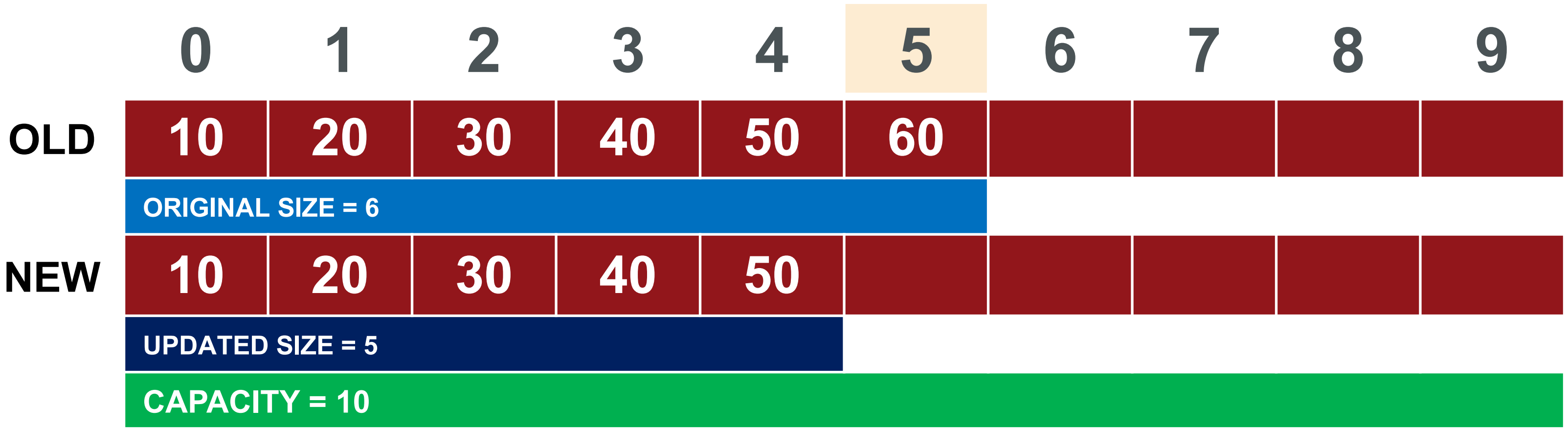
PARTIALLY FILLED ARRAY: POP_BACK

Concept remove the **last** value in the array

Example

```
void pop_back(int &size) {                // size is pass by reference
        if(size > 0) {                    // if array is not empty
                --size;                   // decrement size
        }
}

pop_back(size);                           // remove 60 from the array
```



	0	1	2	3	4	5	6	7	8	9
OLD	10	20	30	40	50	60				
NEW	10	20	30	40	50					

ORIGINAL SIZE = 6

UPDATED SIZE = 5

CAPACITY = 10

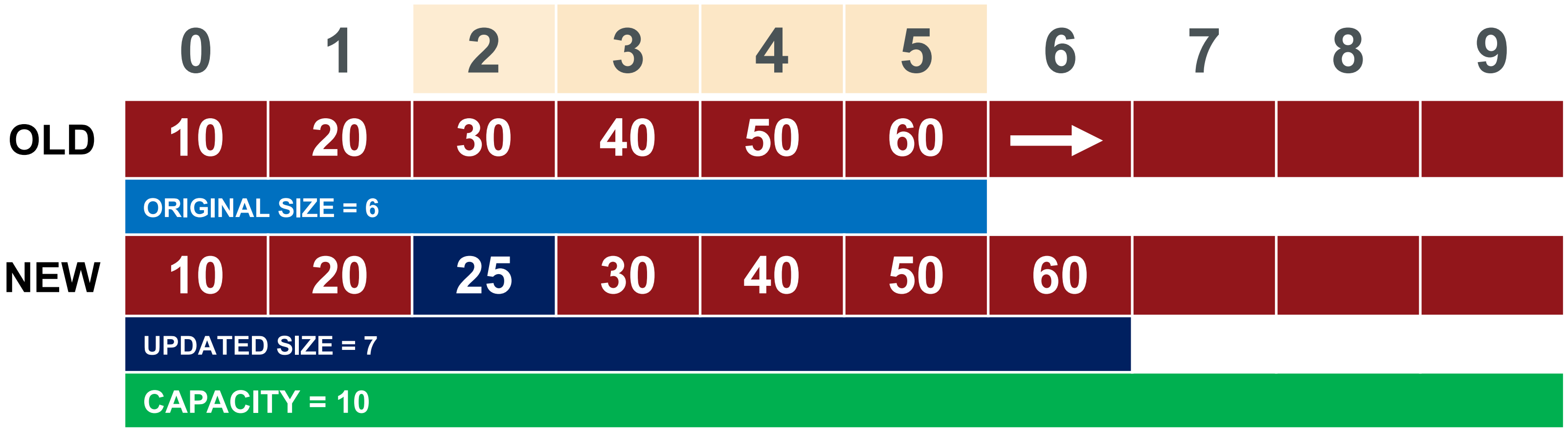
PARTIALLY FILLED ARRAY: INSERT

Concept

insert a value at a specific index in the array

Example

```
void insert( int *a, int capacity, int &size, int value, int index ) {      // size is pass by reference
        if( size < capacity && index >= 0 && index <= size ) {         // if there is room and index is legal
                for(int i=size-1; i>=index; --i) {                   //      shift right ( iterate from 5 to 2 )
                        a[i+1] = a[i];                               //      copy current to next
                }
                a[index] = value;                                    // store value at index
                ++ size;                                             // increment size
        }
}
insert(a, CAPACITY, size, 2, 25);                                    // insert 25 at index 2
```



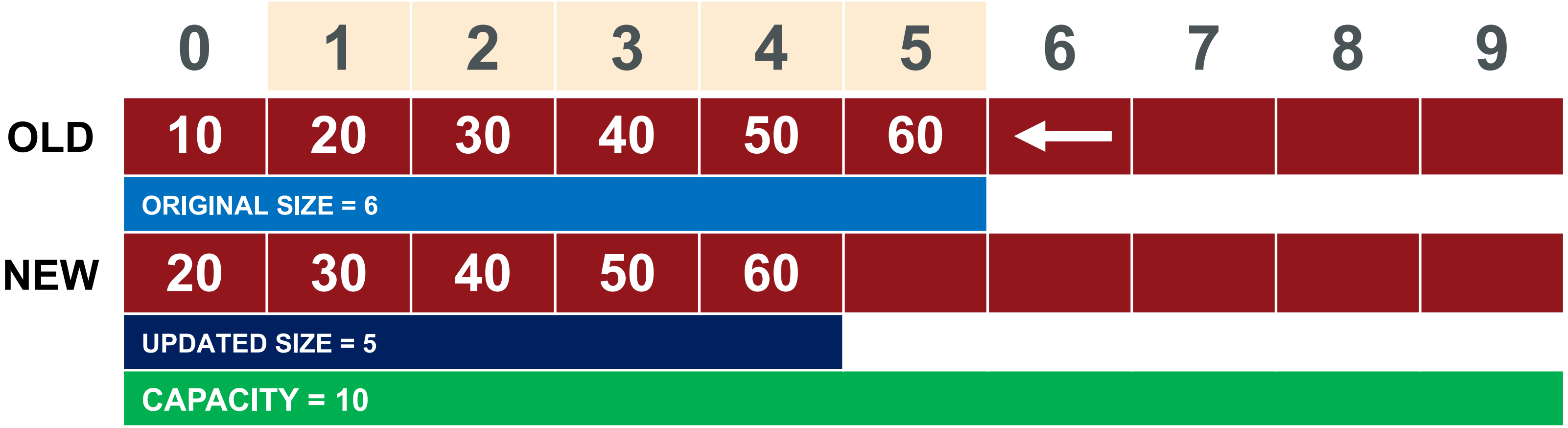
	0	1	2	3	4	5	6	7	8	9
OLD	10	20	30	40	50	60	→			
	ORIGINAL SIZE = 6									
NEW	10	20	25	30	40	50	60			
	UPDATED SIZE = 7									
	CAPACITY = 10									

PARTIALLY FILLED ARRAY: ERASE

Concept insert a value at a specific index in the array

Example

```
void erase( int *a, int capacity, int &size, int index ) {      // size is pass by reference
    if( size > 0 && index >= 0 && index < size ) {              // if there is room and index is legal
        for(int i=index+1; i<size; ++i) {                       //     shift left ( iterate from 1 to 5 )
            a[i-1] = a[i];                                      //     copy current to previous
        }
        --size;                                                 // decrement size
    }
}
erase(a, CAPACITY, size, 0);                                    // erase 10 at index 0
```



	0	1	2	3	4	5	6	7	8	9
OLD	10	20	30	40	50	60	←			
	ORIGINAL SIZE = 6									
NEW	20	30	40	50	60					
	UPDATED SIZE = 5									
	CAPACITY = 10									

DYNAMIC ARRAYS

Concept

arrays stored on the heap using dynamic variables

1. topic is covered as background knowledge for data structures and to increase understanding of memory management

2. easy to make mistakes, difficult to troubleshoot

3. vectors should be used instead of dynamic arrays
   vectors will be covered in ET580

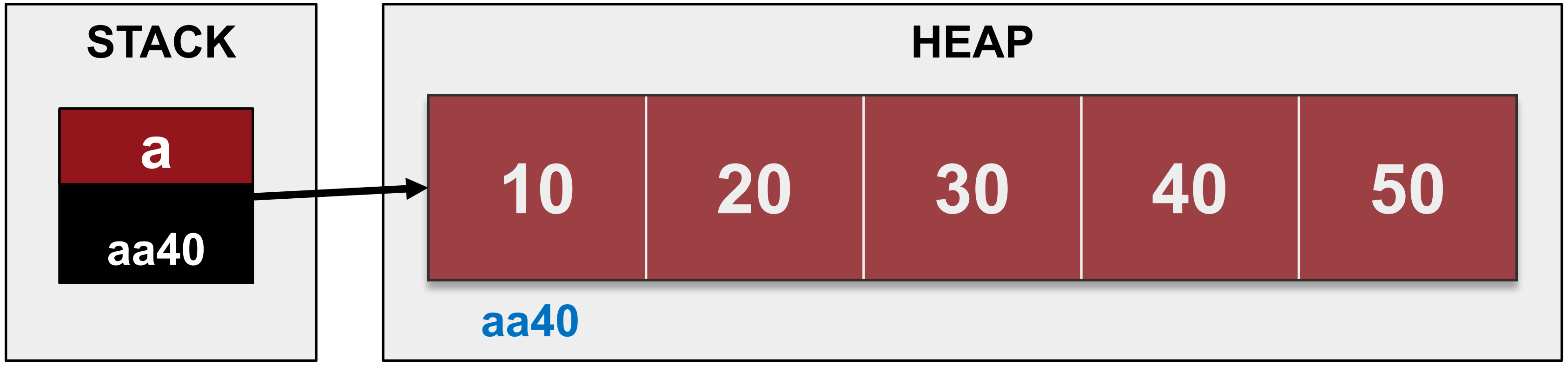
DYNAMIC ARRAYS MEMORY

Concept

an array stored on the heap instead of the stack

```
int *a = new int[5] {10,20,30,40,50};
```

the new operator is required to allocate dynamic memory
a pointer a is required to access this array



PARTIALLY FILLED ARRAY

```
int *a = new int[5] ( );            // array of default integers
```

0	0	0	0	0

```
string *a = new string[5];          // array of empty strings
```

“”	“”	“”	“”	“”

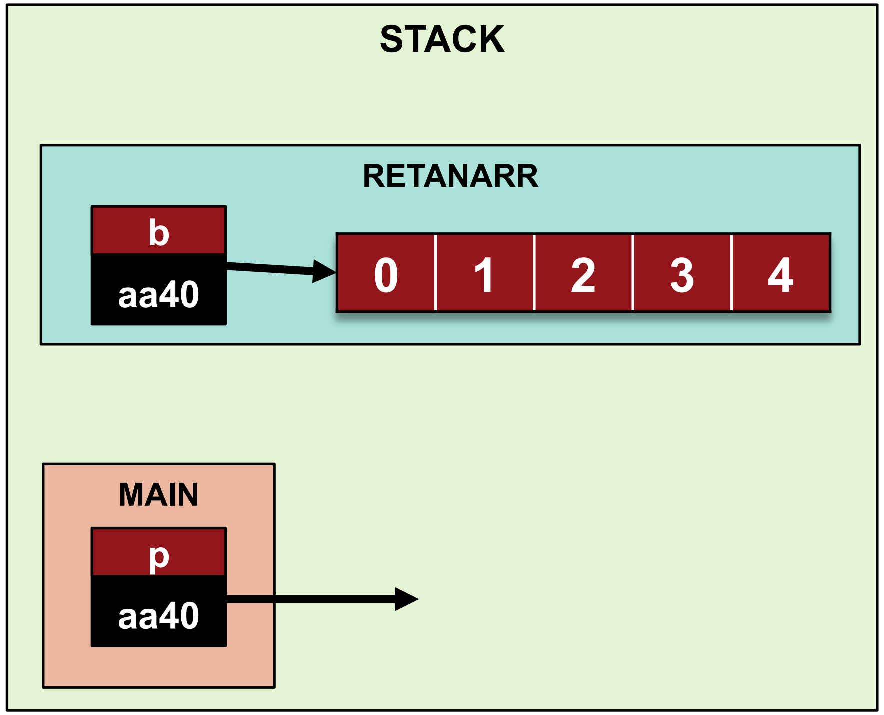
```
int *a = new int[5] {10,20};        // partial initialization
```

10	20	0	0	0

RETURNING A STANDARD ARRAY

```
int* returnAnArray(int size) {
     int b[size];
     for(int i=0; i<size; ++i) { b[i] = i; }
     return b; // array is recycled
}

int main() {
     int size = 5;
     int *p = returnAnArray(size);
}
```

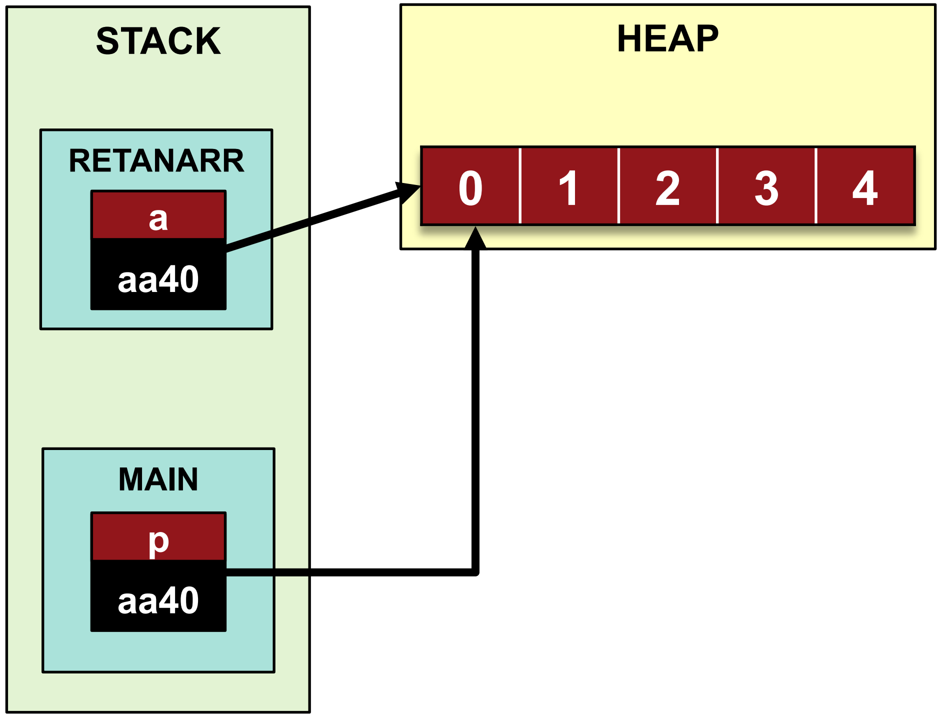


when b goes out of scope the array is recycled, nothing to return

RETURNING A DYNAMIC ARRAY

```
int* returnAnArray(int size) {
    int *a = new int[size];
    for(int i=0; i<size; ++i) { a[i] = i; }
    return a;  // a goes out of scope
}

int main() {
    int size = 5;
    int *p = returnAnArray(size);
}
```



the value of pointer a is stored into p so array remains accessible

ARRAY COMPARISON

standard array

size must be known at compile time (before program runs)
size cannot change during run time (while program runs)

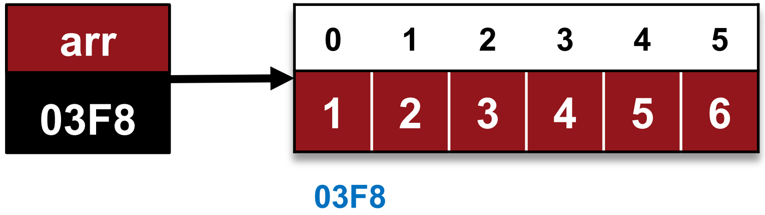
dynamic array

size can be decided during run time
size can be modified (grow or shrink) during run time

DYNAMIC ARRAY: TWO-DIMENSIONAL

```
int rows=2, cols=3;
int *a = new int[rows*cols];

int value = 1;
for(int i=0; i<arrays; ++i) {
     for(int j=0; j<cols; ++j) {
        a[i * cols + j] = value++;       // store values 1 to 6
     }
}
```



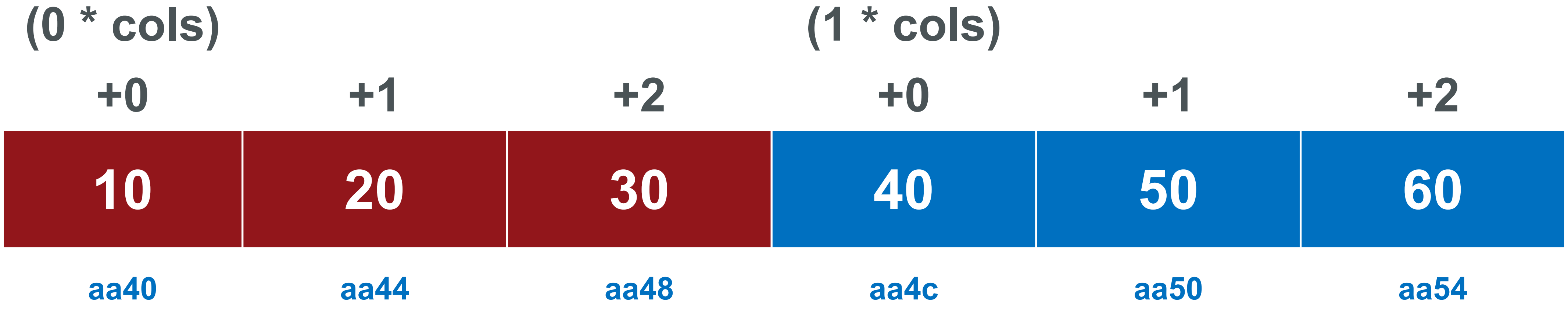
i*cols+j **row major** indexing: 0*3 + 0, 0*3 + 1, 0*3+2, 1*3+0, 1*3+1, 1*3+2

DYNAMIC TWO-DIMENSIONAL ARRAY ARITHMETIC

```
int rows=2, cols=3;
int *a = new int[rows*cols];          // allocate a contiguous row x col block

cout << a[0][1];                      // print 1st array 2nd value
cout << a[1][2];                      // print 2nd array 3rd value

cout << *(p+(0 * cols + 1));          // print 1st array 2nd value
cout << *(p+(1 * cols + 2));          // print 2nd array 3rd value
```



(0 * cols)			(1 * cols)		
+0	+1	+2	+0	+1	+2
10	20	30	40	50	60
aa40	aa44	aa48	aa4c	aa50	aa54