DYNAMIC ARRAYS

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 during templates

STATIC ARRAYS

Static arrays

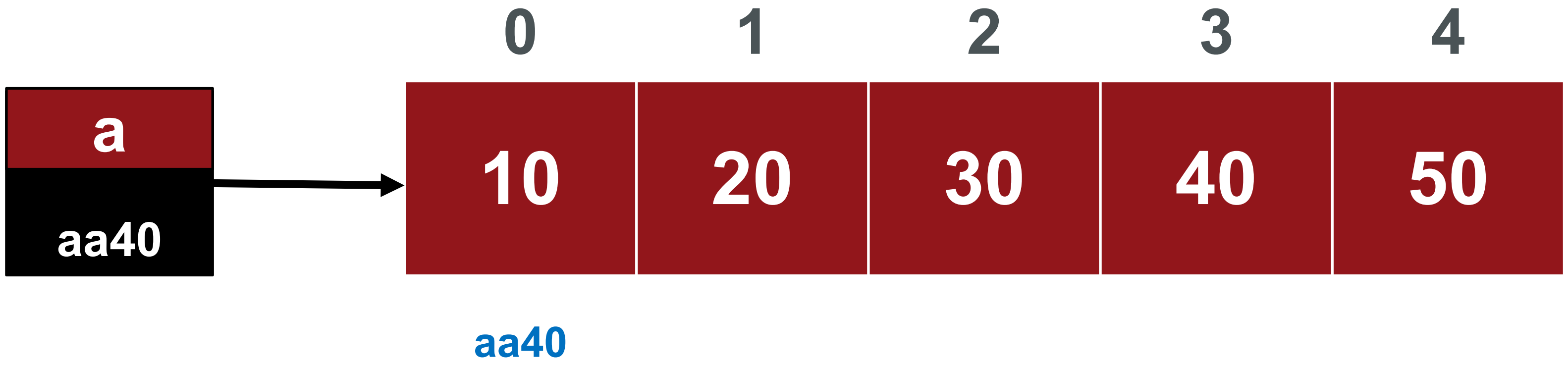
arrays stored on the stack using automatic variables

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

a is an integer array

a functions as a pointer to the first element a[0]

a has a type of int[5] which is an integer array of size 5

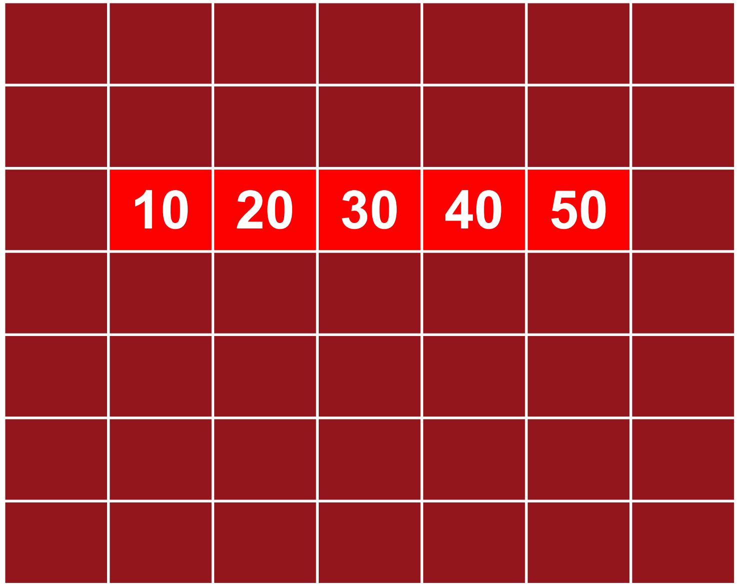


MEMORY CONTIGUITY

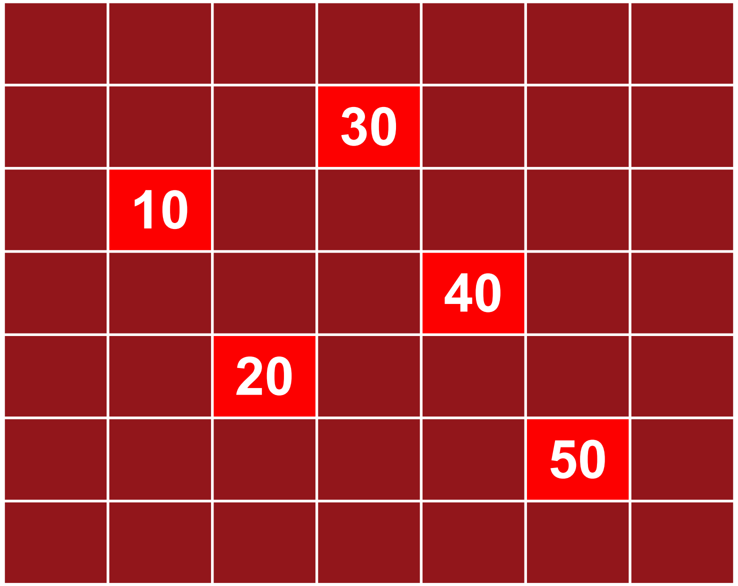
Contiguity a block of adjacent memory cells

an array is a **contiguous** block of memory
each value is stored next to the previous value

Contiguous Block



Non-Contiguous Block

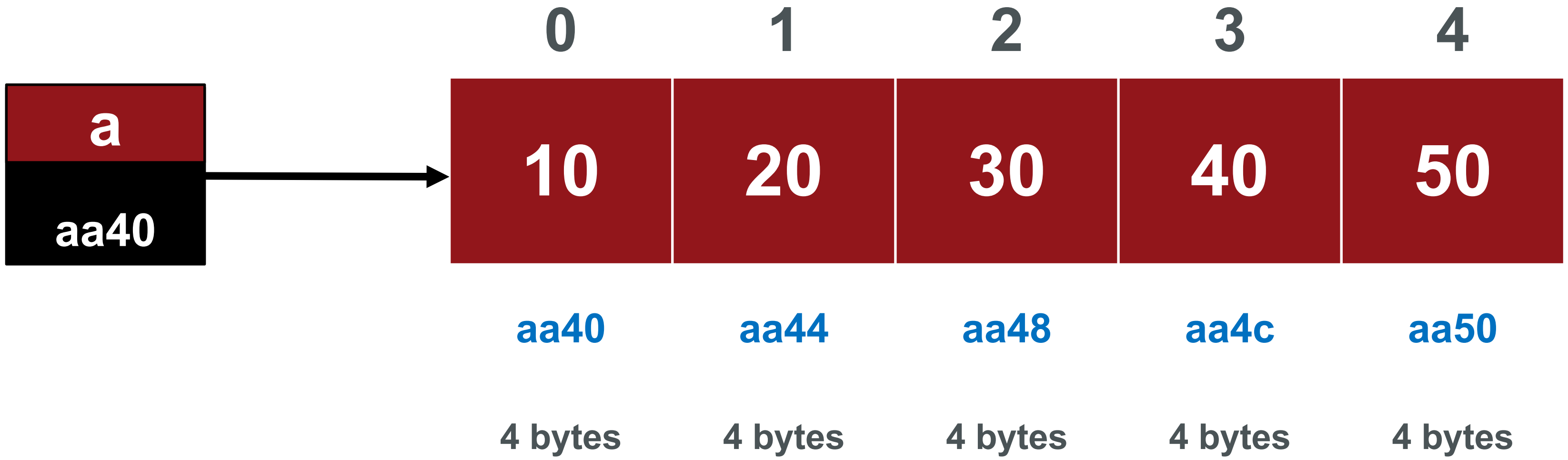


ex2_static_array_contiguity.cpp

MEMORY CONTIGUITY

memory addresses in a contiguous block are "variable size" bytes apart

an integer array stores 4-byte integers in a row
therefore, each memory address should be 4-bytes apart

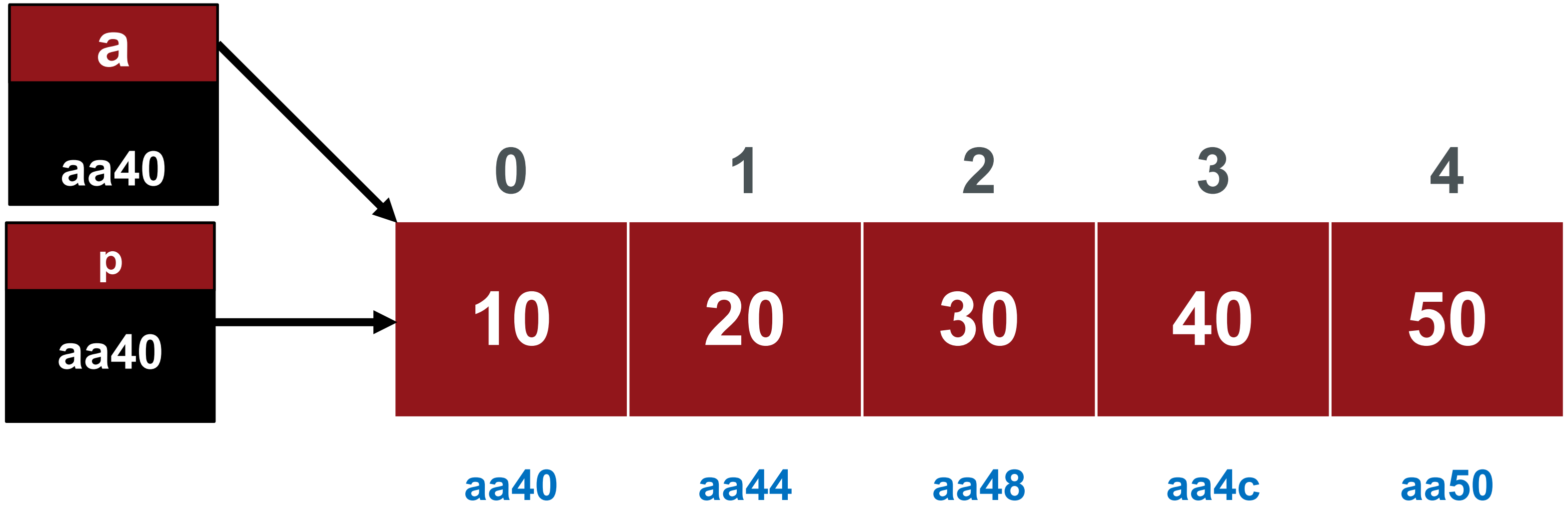


ex2_static_array_contiguity.cpp

POINTERS AND ARRAYS

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

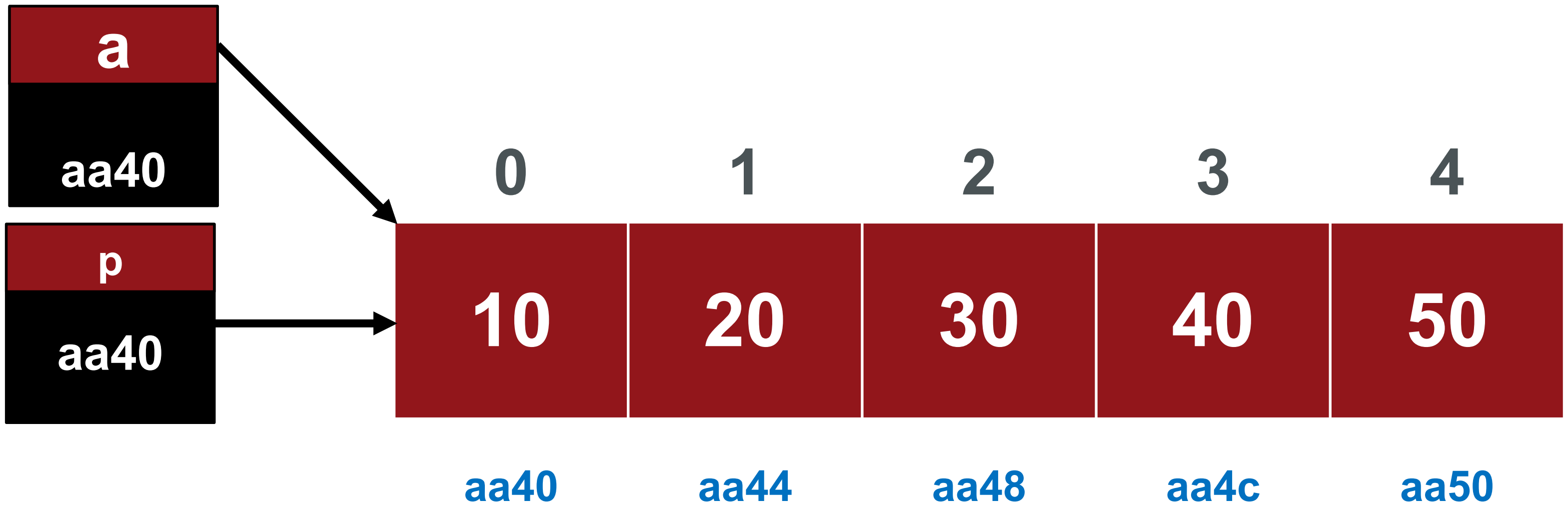
a is of type int[5] which is an integer array of size 5
p is of type int* which can point to any element of an integer array



POINTERS AND ARRAYS

```
int a[5] = {10, 20, 30, 40, 50};
int *p = a;
cout << a[0] << " " << a[1] << "\n";        // print 10 20
cout << p[0] << " " << p[1] << "\n";        // print 10 20
```

[] operator can be used for pointers just like for arrays



TYPE DECAY

Concept

a variables type automatically converts to another type

equivalent function declarations:

```
void print(int b[], int size);
void print(int *b, int size);
```

array parameters (int b[]) decay into pointers (int *b)

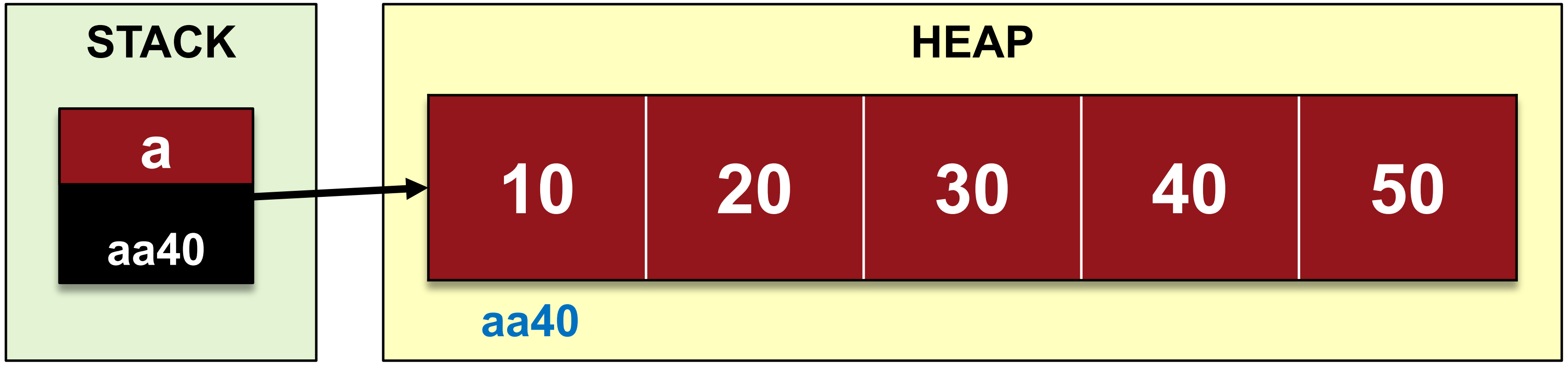
this can be confirmed by checking array variable with sizeof( )

DYNAMIC ARRAY

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



ex4_dynamic_array.cpp

INITIALIZE AN 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
```

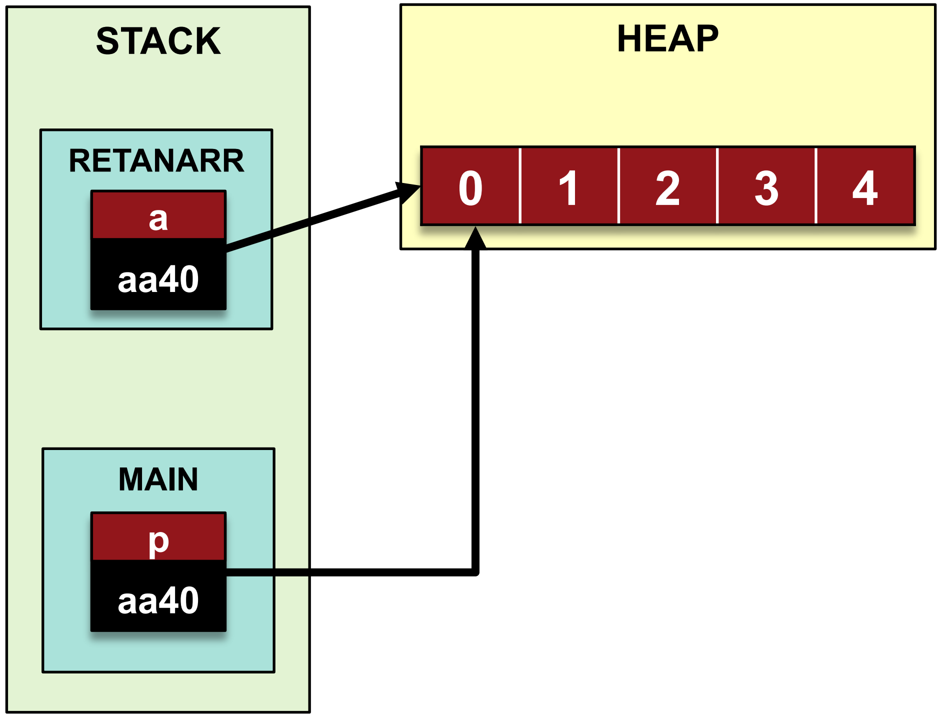
10	20	0	0	0

ex5_array_initialize.cpp

RETURN A LOCAL 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);
}
```



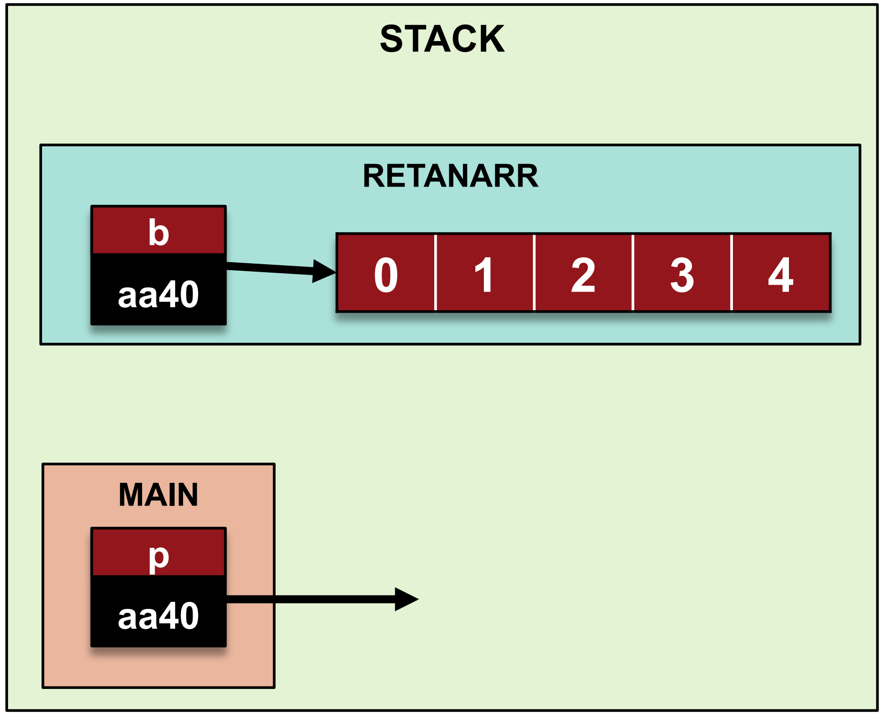
the value of pointer a is stored into p so array remains accessible

ex6a_return_an_array.cpp

RETURN A LOCAL STATIC 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);
}
```

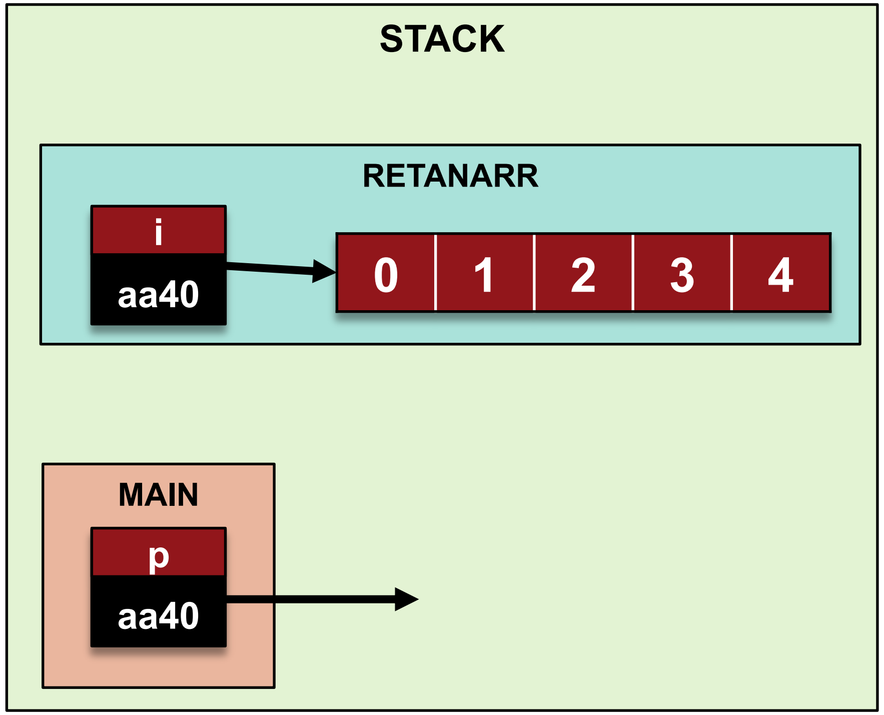


when b goes out of scope the array is recycled, nothing to return

RETURN A LOCAL STATIC 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

DYNAMIC VS STATIC: ARRAY SIZE

static array

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

dynamic array

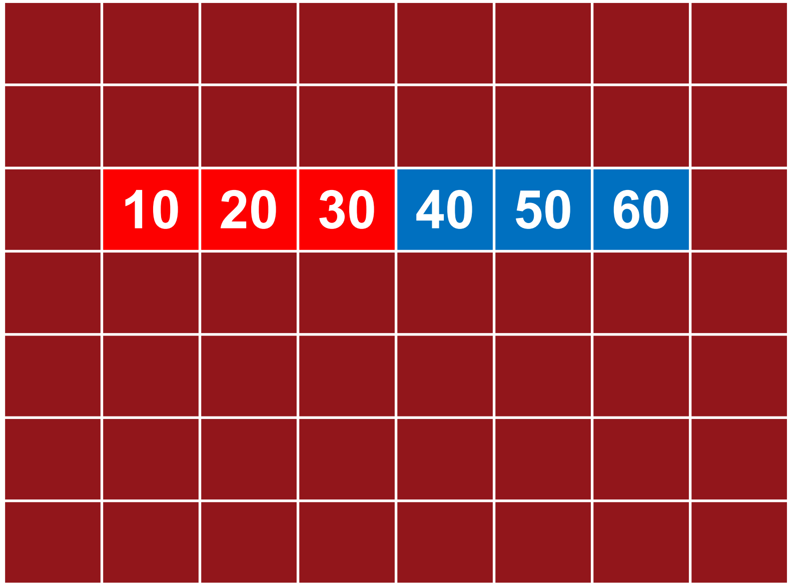
size can be decided during run time
size can be modified (grow or shrink) during run time

ex7_dynamic_array_flexibility.cpp

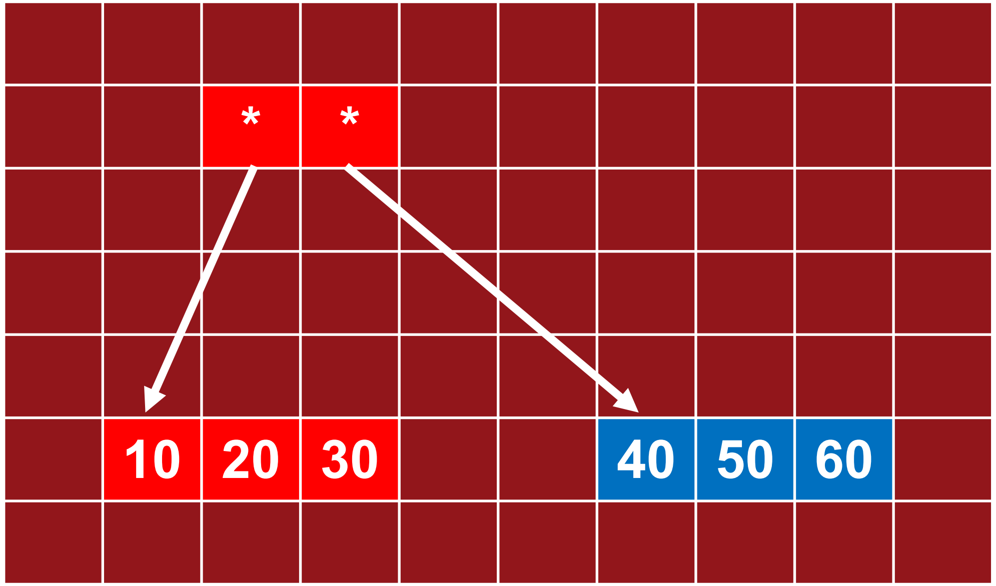
DYNAMIC ARRAY OF ARRAYS

two-dimensional array	a contiguous block of related data
array of arrays	an array of pointers to contiguous arrays

two-dimensional array



array of arrays



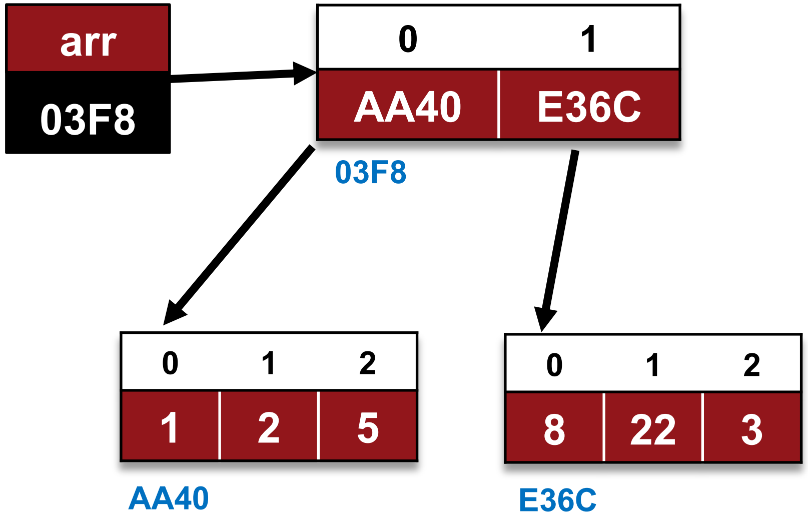
ex8_array_of_arrays.cpp

DYNAMIC ARRAY OF ARRAYS: CREATE

step 1 create an array of pointers
step 2 point each pointer to a new array on the heap

```
int arrays = 2;
int integers = 3;
int **arr = new int*[arrays];        // step 1

for(int i=0; i<arrays; ++i) {        // step 2
    arr[i] = new int[integers];
}
```



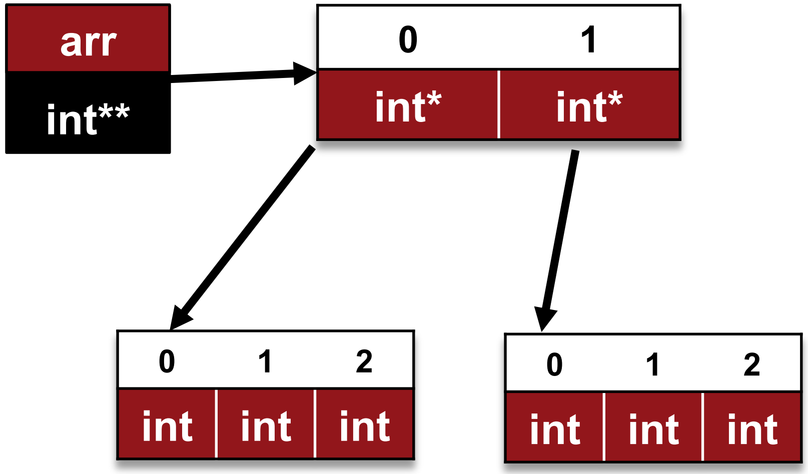
int** is a pointer to an int* pointer or to an int array

DYNAMIC ARRAY OF ARRAYS: DELETE

step 1 delete each integer array
step 2 delete the array of integer pointers

```
for(int i=0; i<arrays; ++i) {          // step 1
     delete [] arr[i];
}

delete [] arr;                         // step 2
```



POINTER ARITHMETIC

Purpose used to access memory before or after a pointer memory cell
works just the same as p[index] where p is a pointer

```
int a[3] = {10, 20, 30};

cout << a[0];        // prints 10
cout << a[1];        // prints 20

cout << *(a+0);      // prints 10
cout << *(a+1);      // prints 20
```

*(a+x) - add x memory cells to the pointer a then dereference

ex9_pointer_arithmetic.cpp

POINTER ARITHMETIC: ARRAY OF ARRAYS

Concept shift left or right from a pointer’s location by pointer size bytes

```
cout << a[0][1];          // print first array second value
cout << a[1][3];          // print second array third value

cout << *(*(a+0)+1);      // print first array second value
cout << *(*(a+1)+3);      // print second array third value
```

ex11_pointer_arithmetic.cpp