CS 11: Introduction to Computer Science

Pointers, The Heap and Dynamic List Arrays

Mark Sheldon
Tufts University
Email: msheldon@cs.tufts.edu
Web: http://www.cs.tufts.edu/~msheldon

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah

Goals for this session

- **Review pointers & arrays**
- **Review using "new" and allocating space on the heap**
- **Applying what we've learned: dynamic array lists**
- **Preparing to explore object-oriented programming**

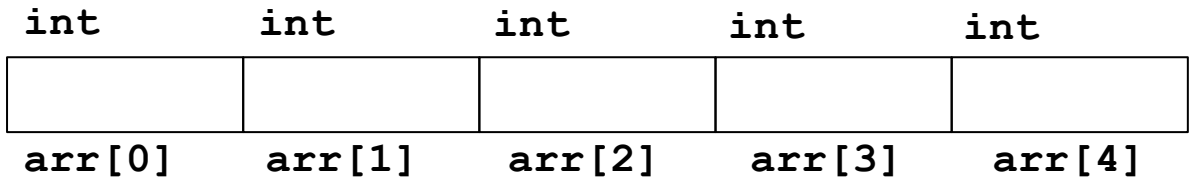
Unscrambling Pointers and Arrays

What you should get from this interlude

- **Often C++ will take the address of an array for you when a pointer is needed**
- **In C++ you will see subscripting used on pointers (to arrays) as well as on arrays. *In all cases it means the same thing.***
- **You will see `arr[]` (especially in parameter declarations) and `*arr_ptr` syntax used somewhat interchangeably to declare *pointers to arrays*.**

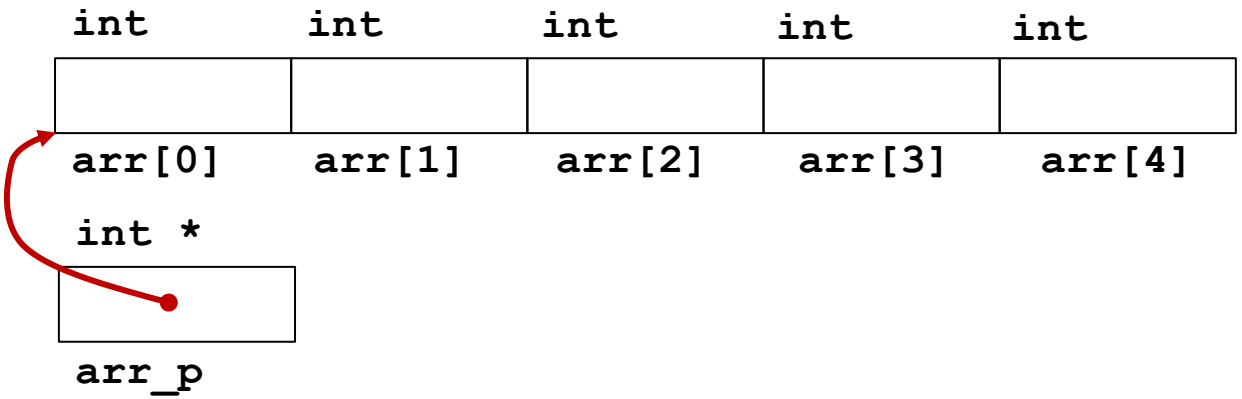
Pointers, Arrays, and how they relate

```
int arr[5];
```



Pointers, Arrays, and how they relate

```
int arr[5];

int *arr_p = &arr;
```

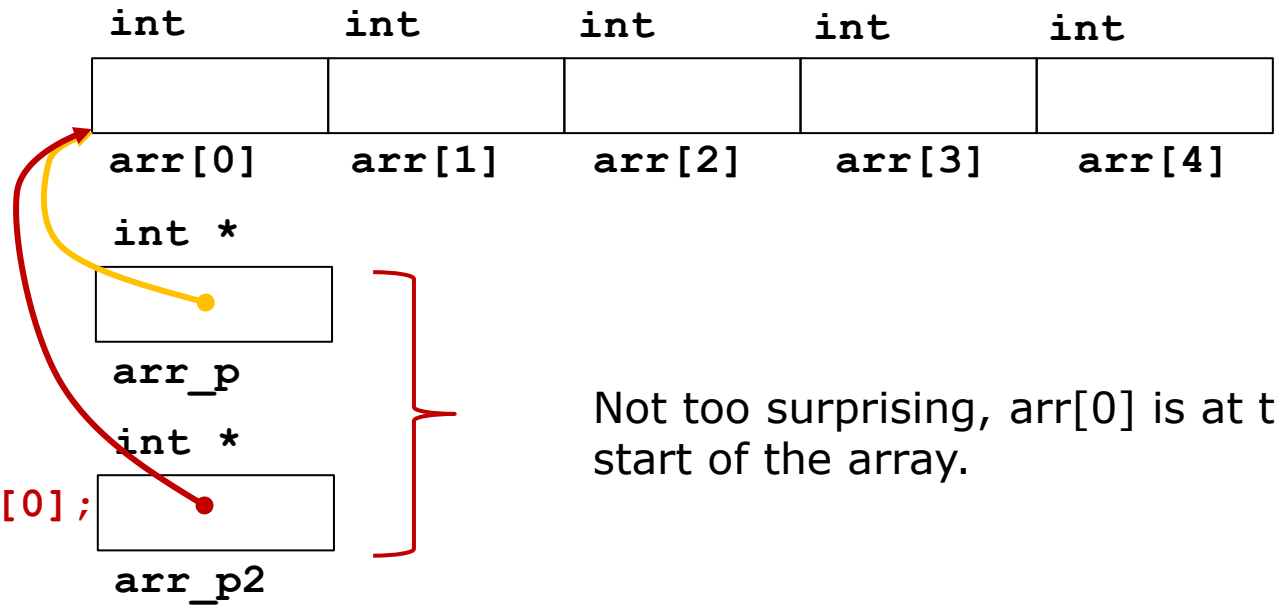


That's pretty clear…

…but C++ for historical reasons plays some tricks. You'll need to at least not be surprised when you see the following….

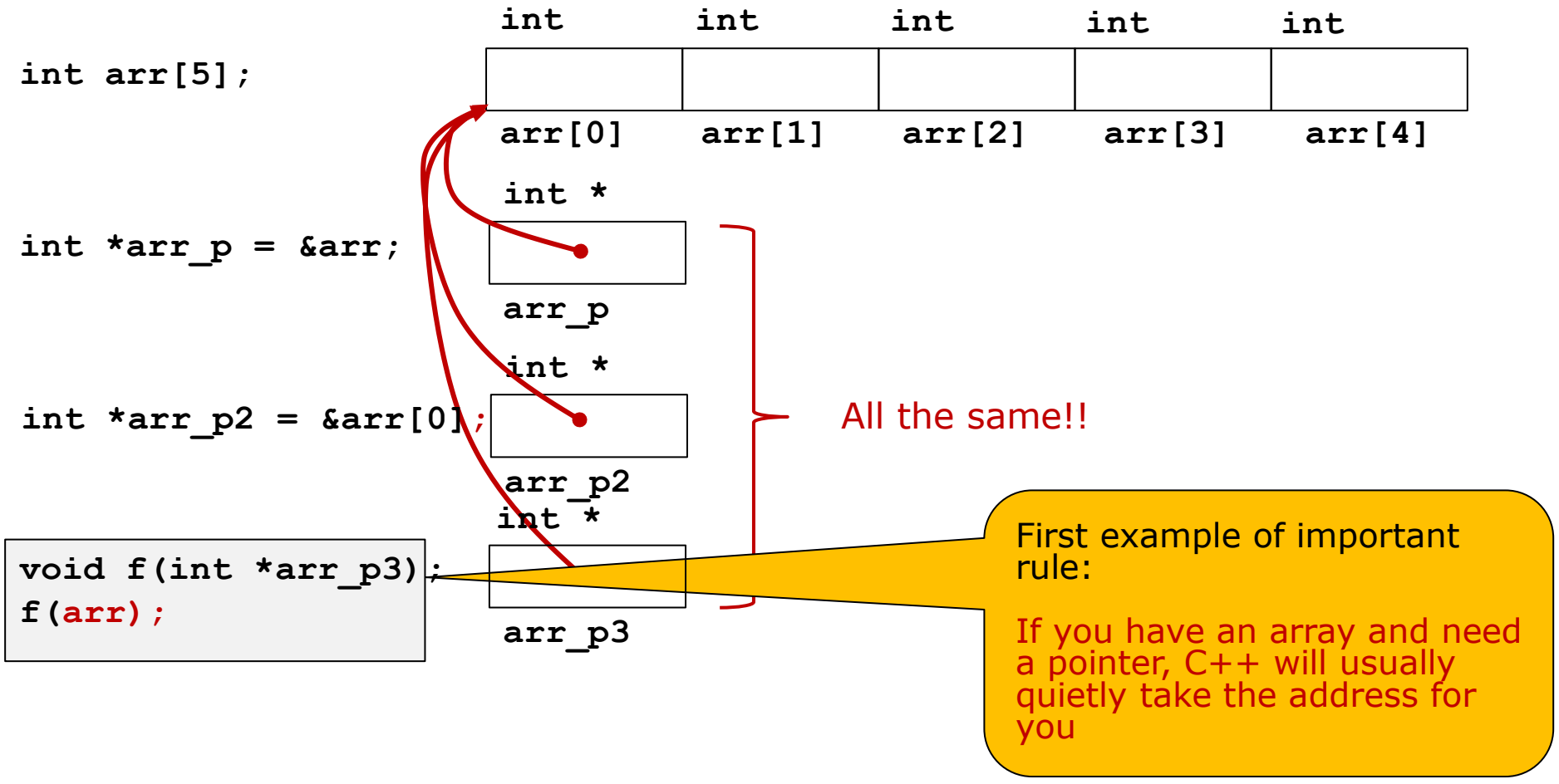
&array and &array[0] are always the same

```
int arr[5];

int *arr_p = &arr;

int *arr_p2 = &arr[0];
```

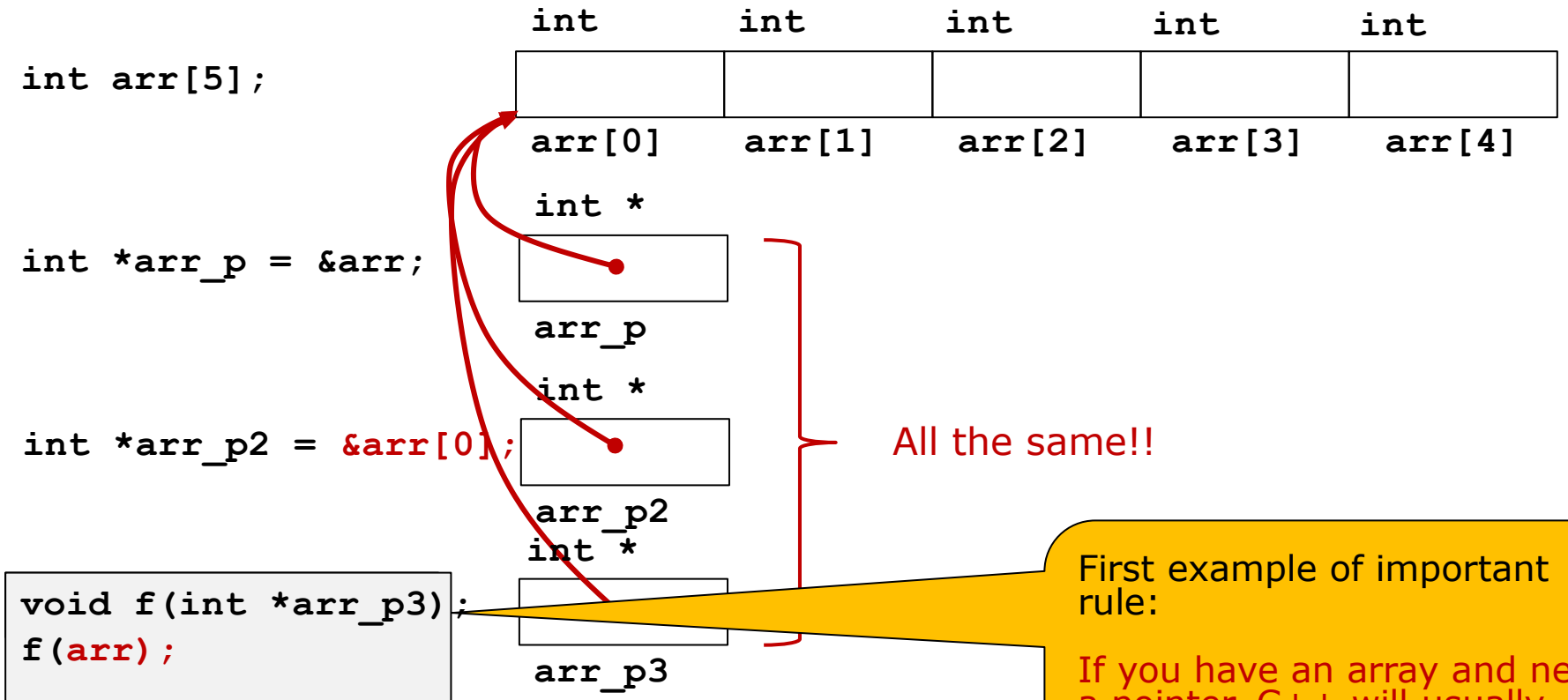


Not too surprising, arr[0] is at the start of the array.

C++ will often take the address of a pointer for you



&array and &array[0] are always the same



That's a lot of detail…what you mostly need to know is…

In many but not all cases, pointer and array types can be used somewhat interchangeably. Most importantly:

```
void f(int *parm);
void f(int parm[]);
```

Truly equivalent function signatures…so get used to seeing both!

And now we need add one more simple part of the story:

You can subscript pointers…*assumed* pointing to array

In many but not all cases, pointer and array types can be used somewhat interchangeably. Most importantly:

```
void f(int parm[]) {
   cout << parm[1];   // works, no surprise it's an array
};
```

```
void f(int *parm) {
   cout << parm[1];   // works exactly the same!!
};
```

In C++

Subscripting an array and subscripting a pointer to that array are *by definition doing exactly the same thing!*

Use whichever is more convenient!

You can subscript pointers…*assumed* pointing to array

In many but not all cases, pointer and array types can be used somewhat interchangeably. Most importantly:

```
void f(int parm[]) {
   cout << parm[1];   // works, no surprise it's an array
};
```

```
void f(int *parm) {
   cout << parm[1];   // works the same!!
};
```

```
int arr = {1,2,3};
int *ar_ptr = &arr;
cout << arr [2];     // prints 3 (which is in arr[2])
cout << ar_ptr[2];   // prints 3 (which is in arr[2])
```

What you should have gotten from this interlude

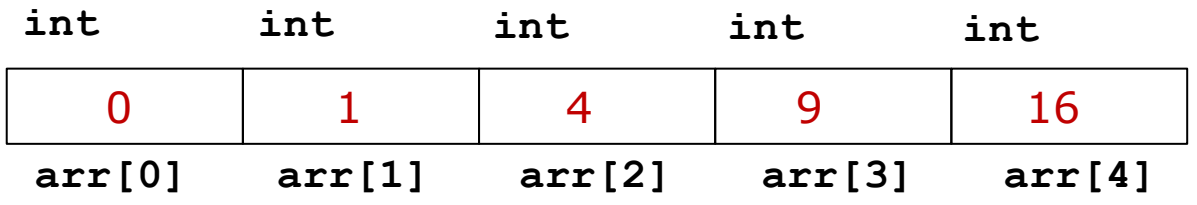
- **Often C++ will take the address of an array for you when a pointer is needed**
- **In C++ you will see subscripting used on pointers (to arrays) as well as on arrays. *In all cases it means the same thing***
- **You will see `arr[]` (especially in parameter declarations) and `*arr_ptr` syntax used somewhat interchangeably to declare pointers to arrays**
- **`arr[5]` with an explicit size is different…that's an array of known size, but…**
- **…as always, all arrays are passed into functions by *reference***

Using **`new`** to get Memory on the Heap
An Example Application

I'd like to write a main program like this:

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```

How many squares do you want to store? 5

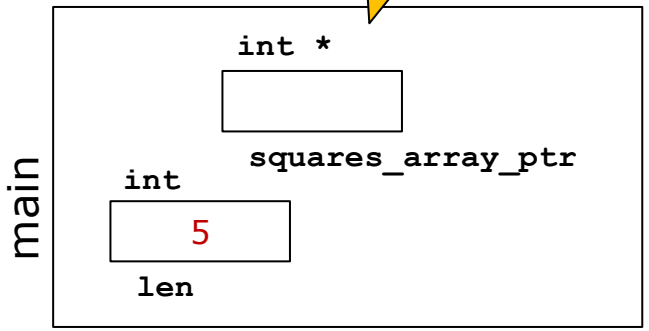


But where do the variables live?

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```

How many squares do you want to store? 5

main()'s local variables live in its activation record, of course.



But where do the variables live?

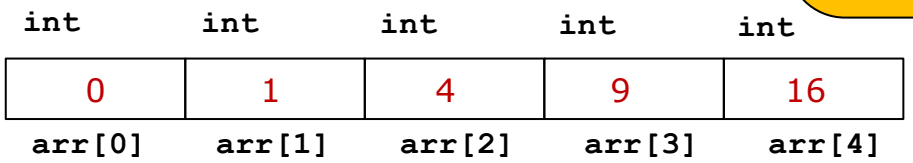
```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squa
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```

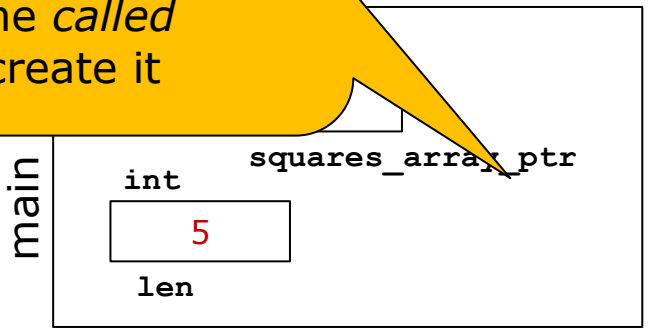
But where can the array live?

Not in main()'s activation record…

…we want the *called* function to create it

How many squares do you want to store? 5



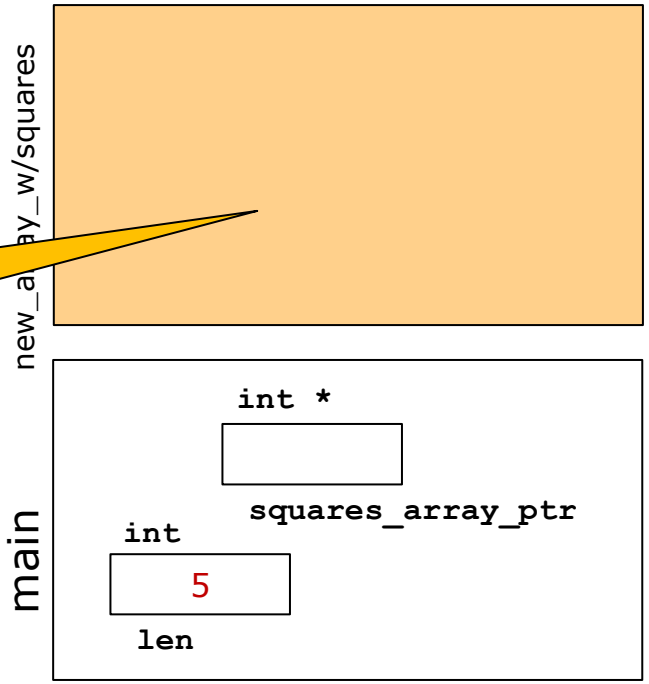


But where do the variables liv

But where can the array live?

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
```

It can't be in the activation record for new_array_with_squares…



But where do the variables liv

But where can the array live?

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
```

It can't be in the activation record for new_array_with_squares…

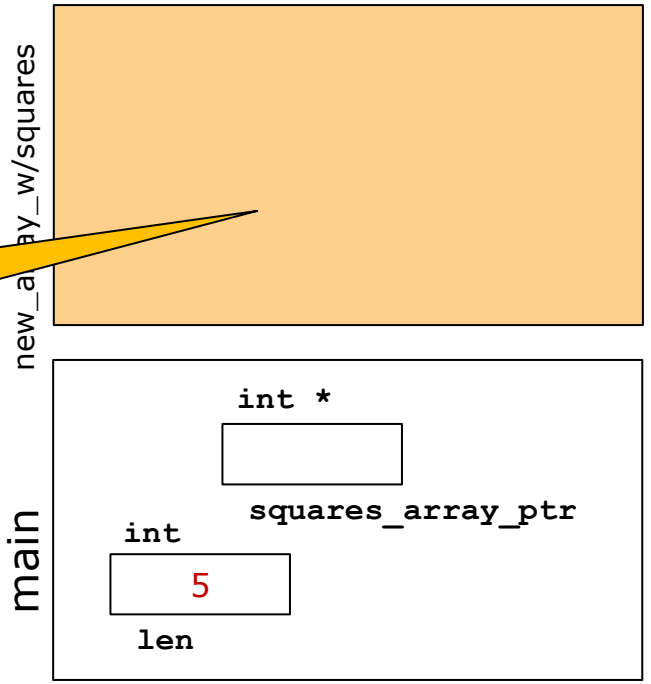
…that will go away when the function returns!

ore? 5

int

16

arr[0] arr[1] arr[2] arr[3] arr[4]

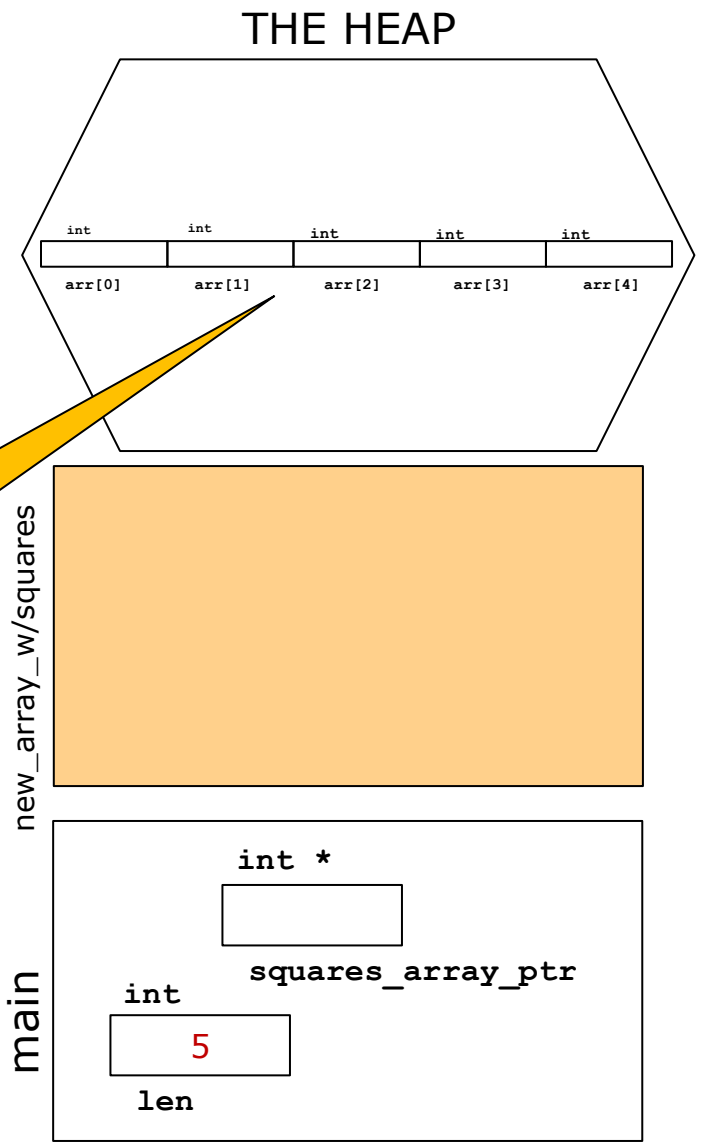


But where do the variables live?

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? "
        cin  >> len;

        squares_array_ptr = new_array_with_squ      en);
        print_array(squares_array_ptr, len
        delete [] squares_array_ptr;
}
```

5

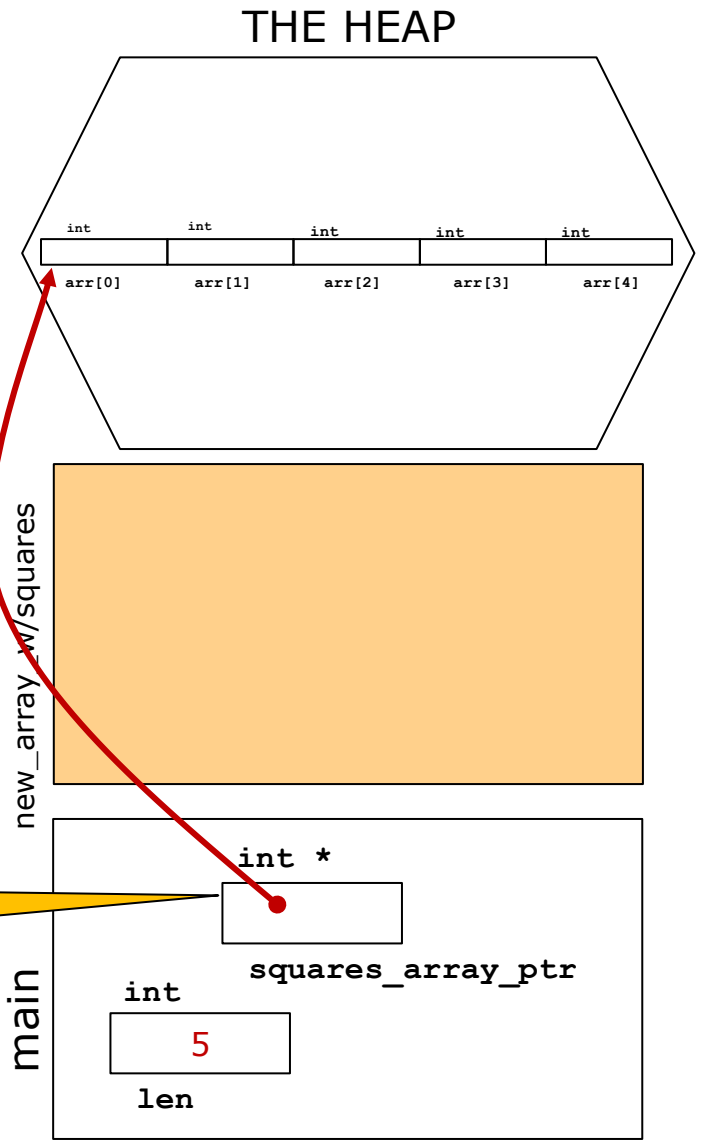
new_array_with_squares will allocate it on the heap using new…



But where do the variables live?

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```

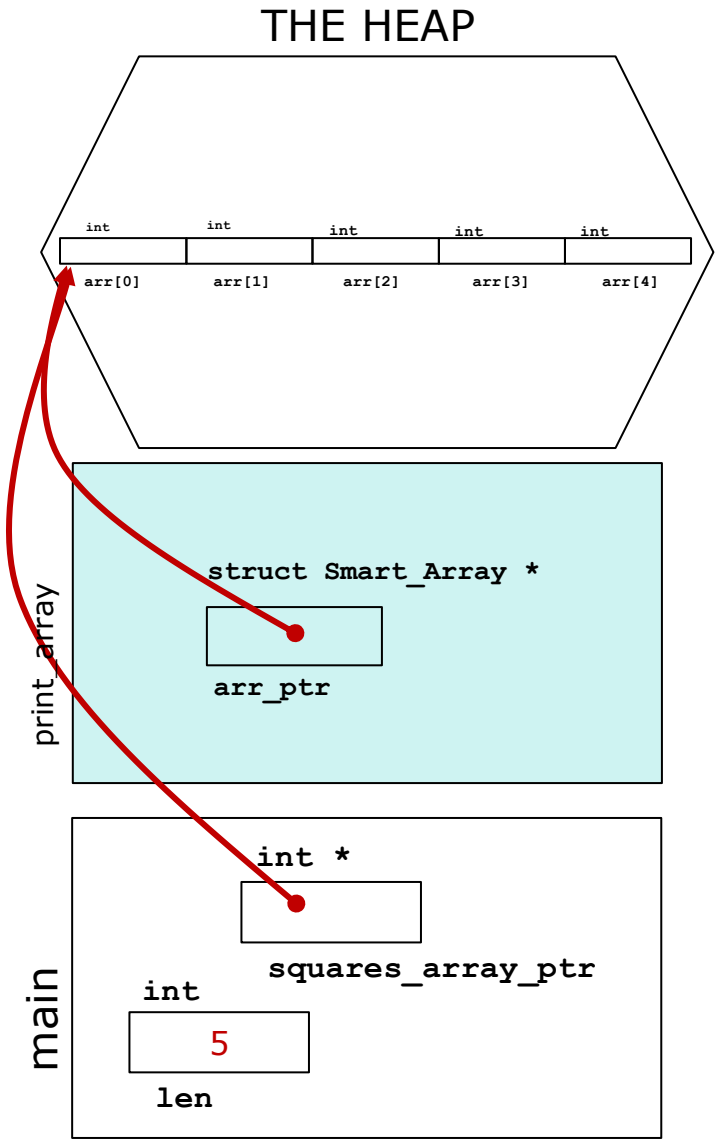
…and will return a pointer that we can keep here.



But where do the variables live?

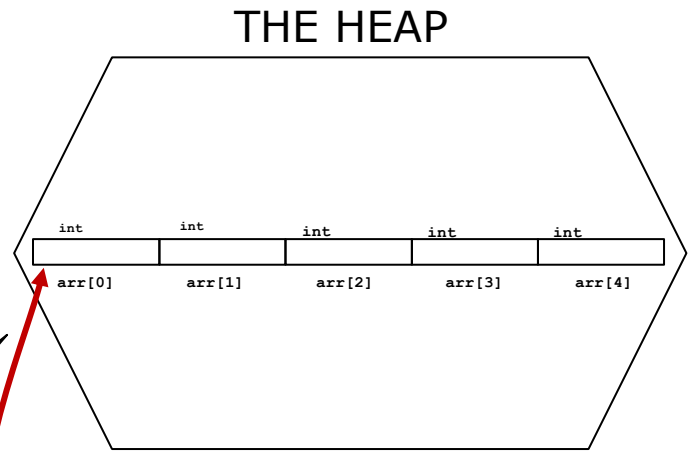
```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```

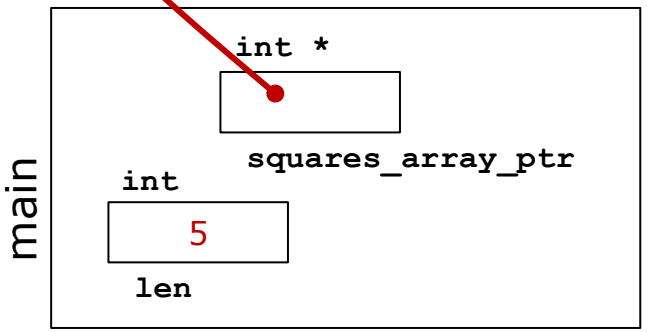
…so we can pass the pointer into the printing function which can then find the array!



But where do the variables live?

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to sto      ";
        cin  >> len;

        squares_array_ptr = new_array_with_sq        len);
        print_array(squares_array_ptr, len
        delete [] squares_array_ptr;
}
```

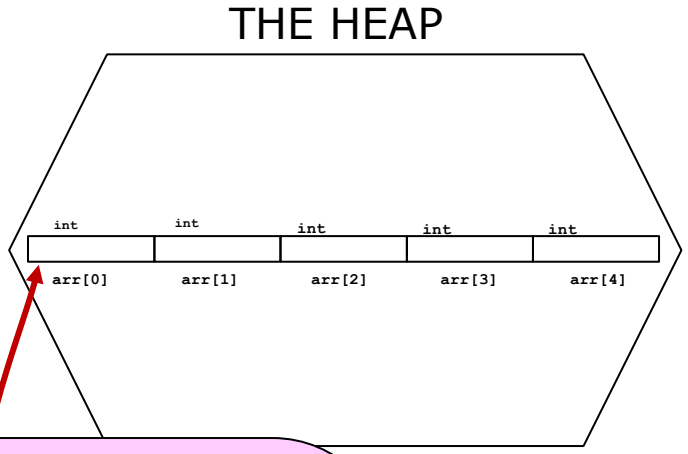




Crucially…

Both the array and at least one pointer to it live on after the function that created the array returns!

But where do the variables live?

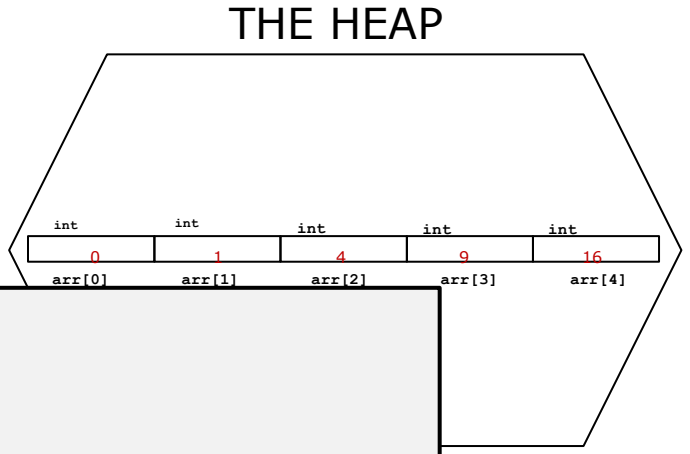



```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int
}
```

```
array_ptr
```

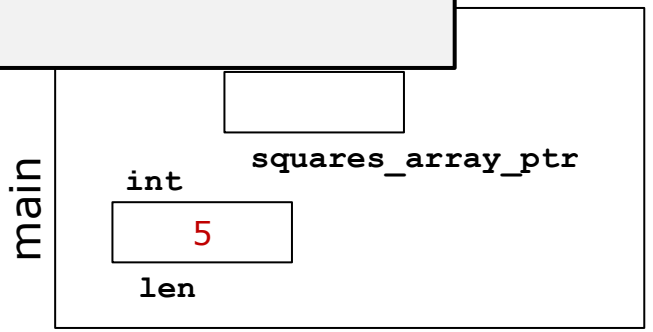
But.. how can `new_array_with_squares` create this array and return the pointer?

Creating the array on the heap



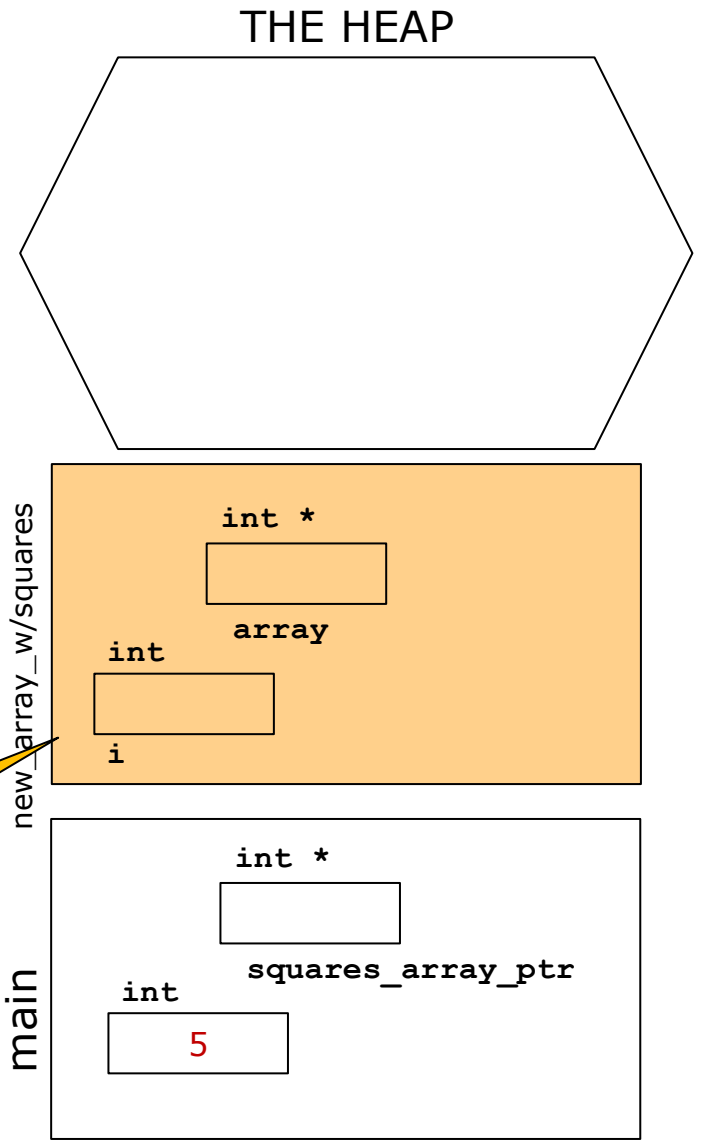

```
int *new_array_with_squares(int size)
{
        int  i;
        int *array = new int [size];

       // Fill with squares
       for (i = 0; i < size; i++) {
                array[i] = i * i;   // array[i] gets i squared
        }
        return array;
}
```

How many squares do you want to store? 5



Creating the array on the heap

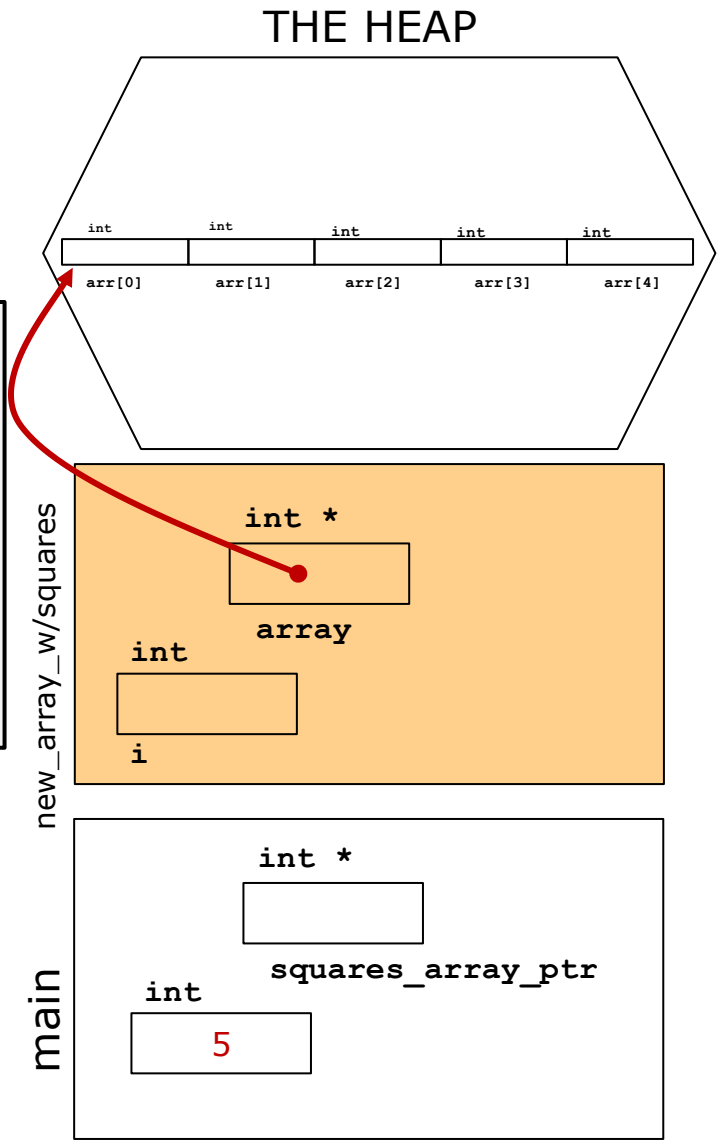
```
int *new_array_with_squares(int size)
{
        int  i;
        int *array = new int [size];

        // Fill with squares
        for (i = 0; i < size; i++) {
                array[i] = i * i;   // array[i] gets i squared
        }
        return array;
}
```



new_array_with_squares' local variables live in its activation record *while it is running!*

Creating the array on the heap

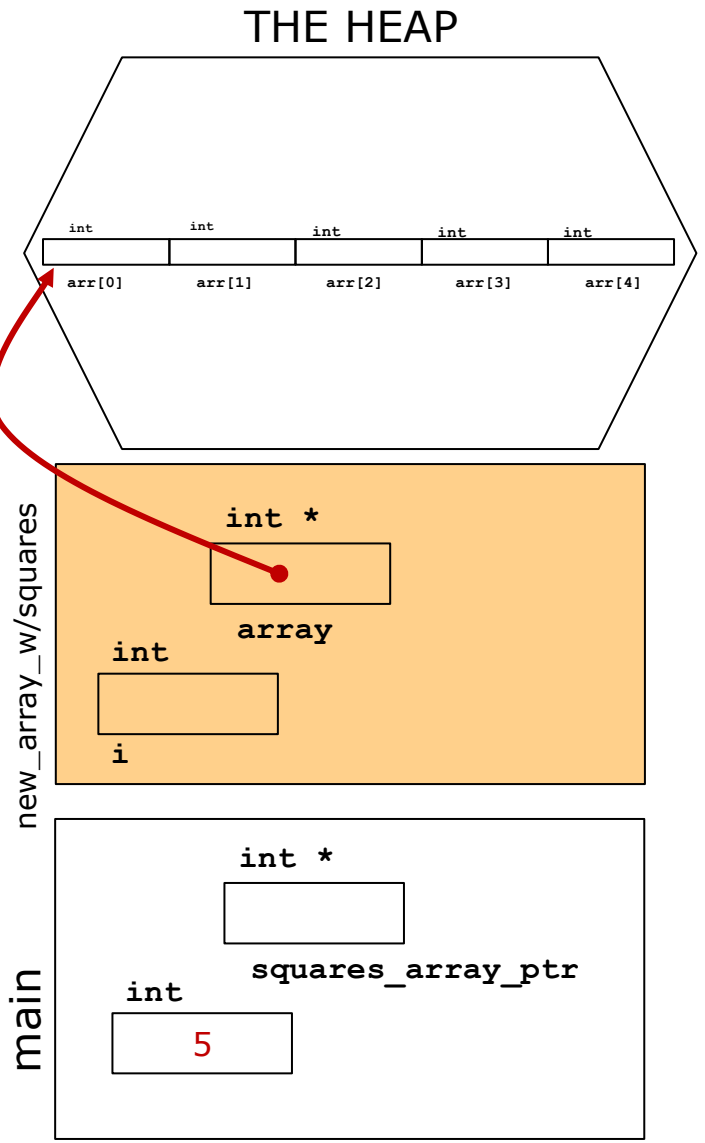
```
int *new_array_with_squares(int size)
{
        int  i;
        int *array = new int [size];

        // Fill with squares
        for (i = 0; i < size; i++) {
                array[i] = i * i;   // array[i] gets i squared
        }
        return array;
}
```



Creating the array on the heap

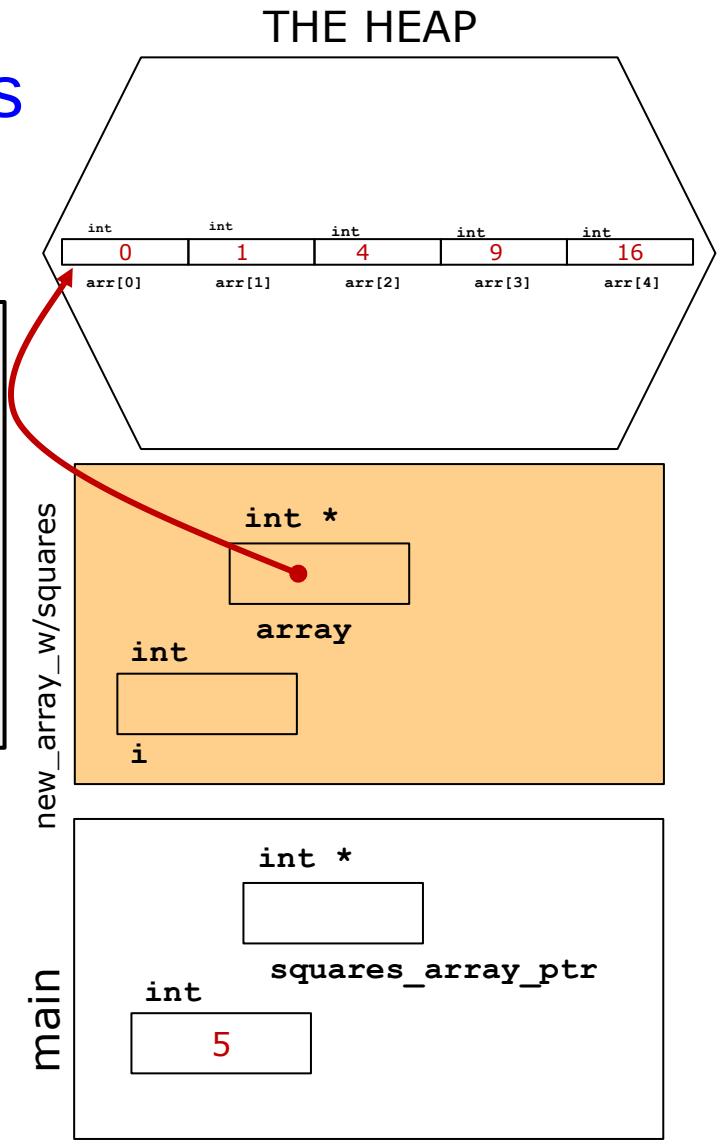
The C++ new operation allocates the array *on the heap!*

```
int *new_array_w                    t size)
{
        int  i;
        int *array = new int [size];

        // Fill with squares
        for (i = 0; i < size; i++) {
                array[i] = i * i;   // array[i] gets i squared
        }
        return array;
}
```



Filling the array with squared numbers

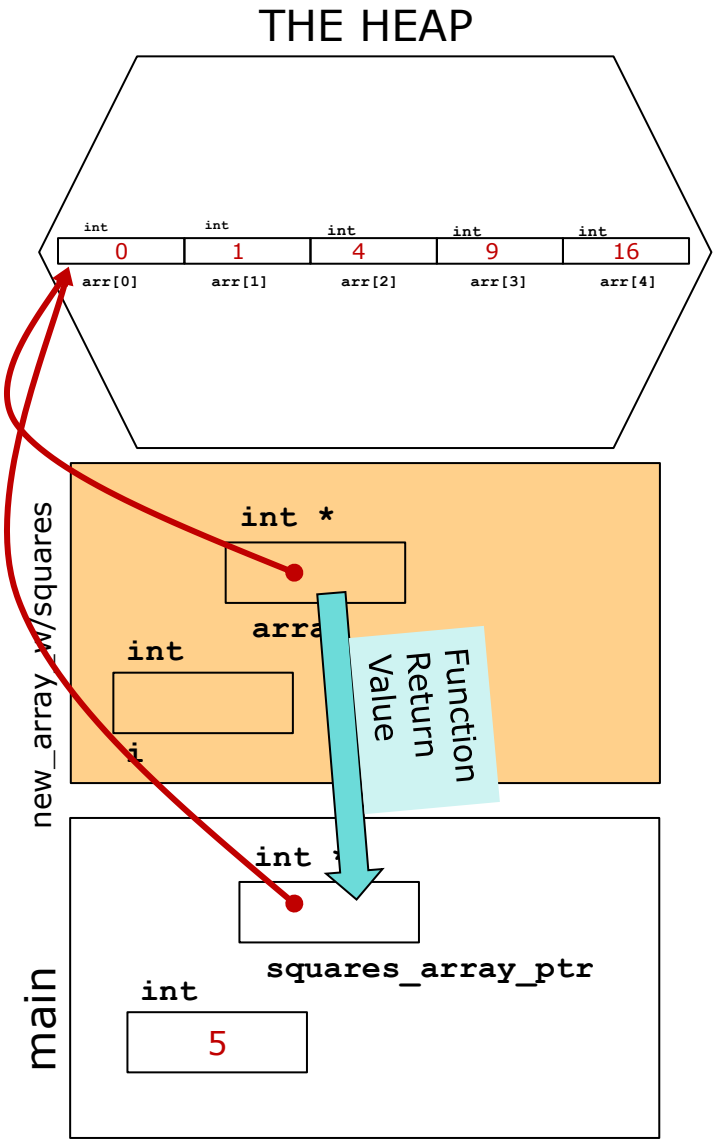
```
int *new_array_with_squares(int size)
{
        int  i;
        int *array = new int [size];

        // Fill with squares
        for (i = 0; i < size; i++) {
                array[i] = i * i;   // array[i] gets i squared
        }
        return array;
}
```



How does main find the array?

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```

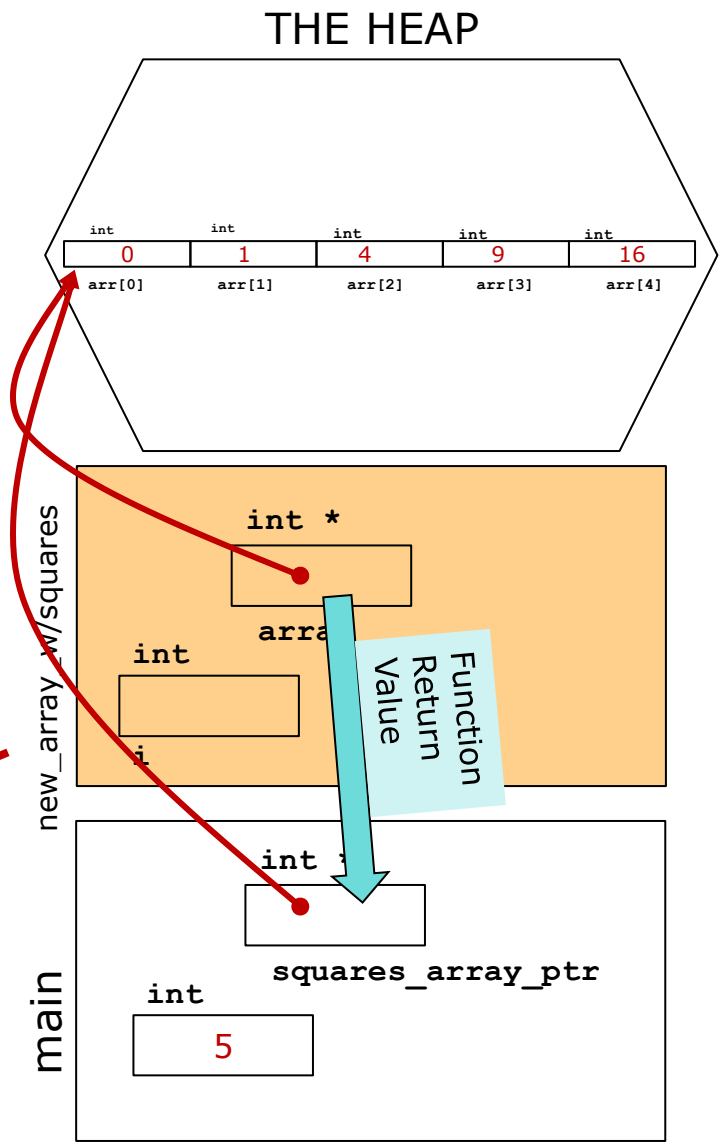
```
int *new_array_with_squares(int size)
{
        int  i;
        int *array = new int [size];

        // Fill with squares
        for (i = 0; i < size; i++) {
                array[i] = i * i;   // array[i] gets i squared
        }
        return array;
}
```



How does main find the array?

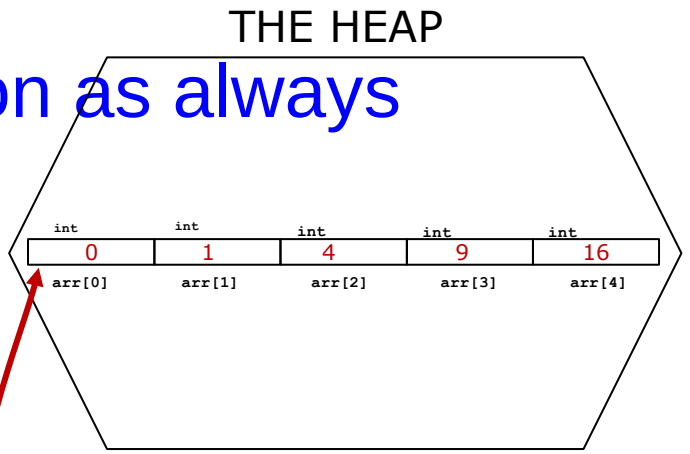
```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```

```
int *new_array_with_squares(int size)
{
        int  i;
        int *array = new int [size];

        // Fill with squares
        for (i = 0; i < size; i++) {
                array[i] = i * i;   // array[i] gets i squared
        }
        return array;
}
```



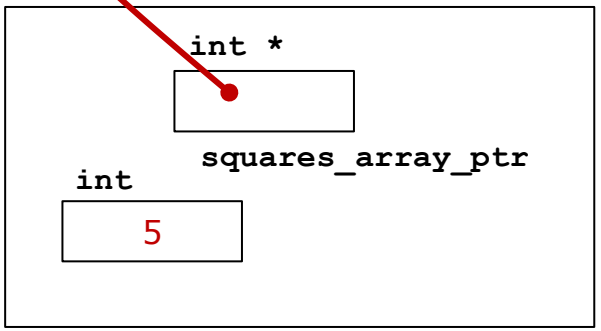
Now we can pass the array to a function as always

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```



How many squares do you want to store? 5

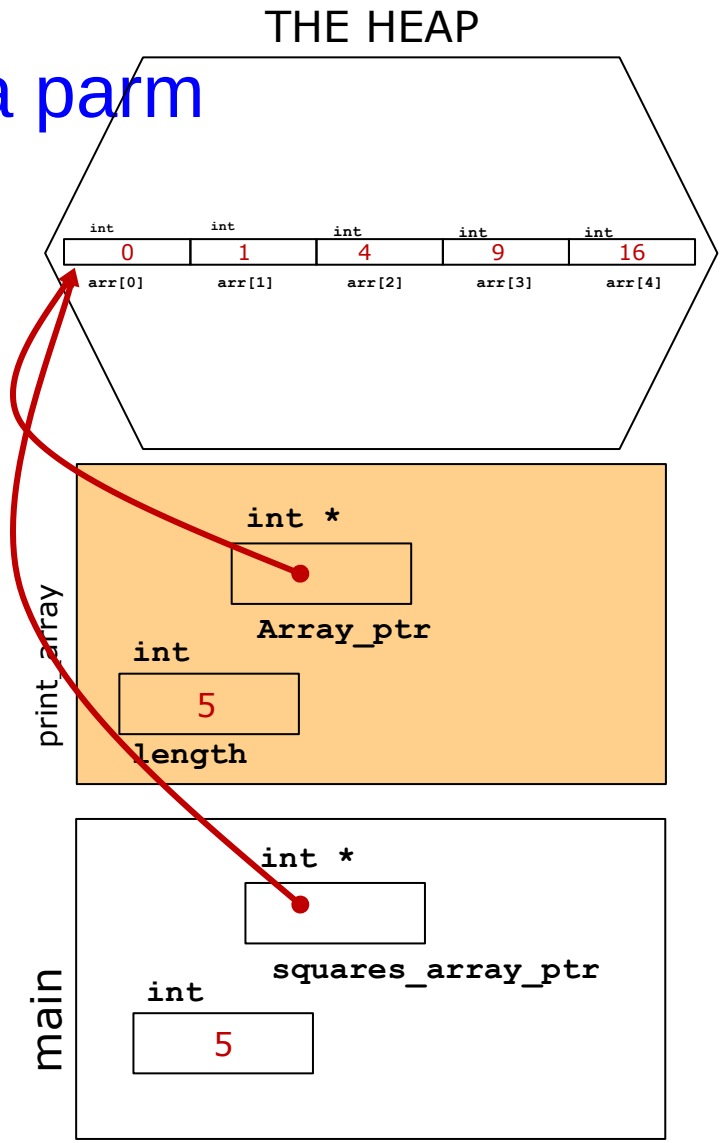
```
array[0] =  0
array[1] =  1
array[2] =  4
array[3] =  9
array[4] =  16
```



Print_array gets the array pointer as a parm

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```

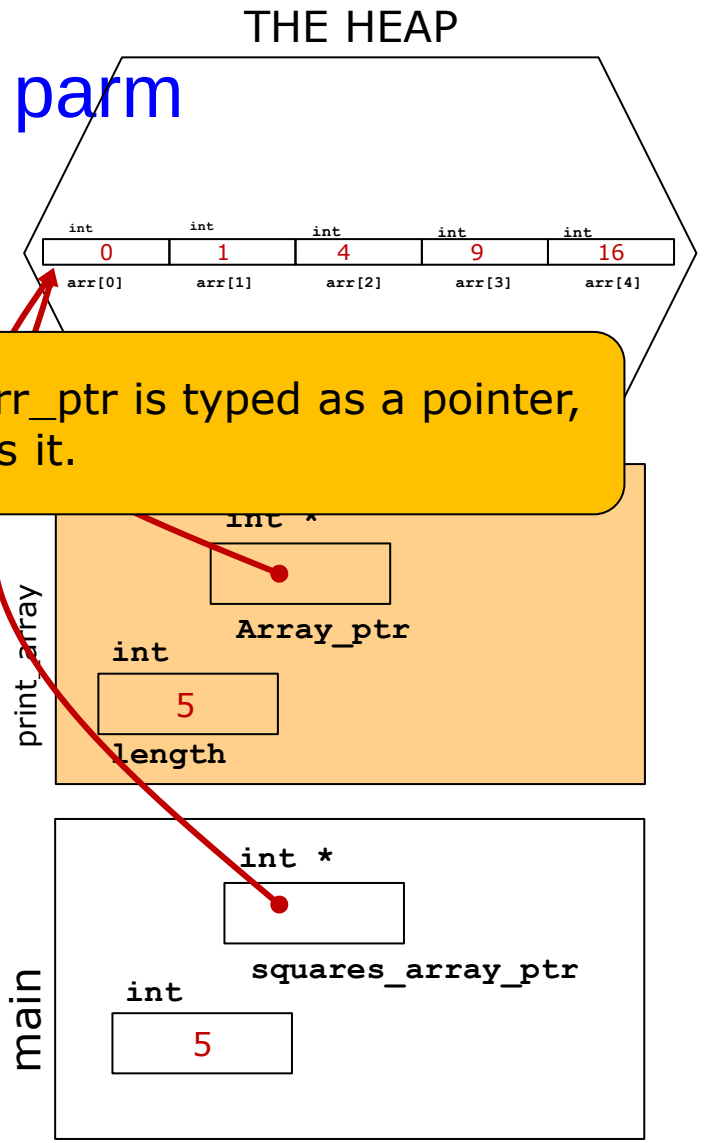
```
void print_array(int *arr_ptr, int length)
{
        int i;

        for (i = 0; i < length; i++) {
                cout << "array[" << i << "] =  "
                     << arr_ptr[i] << endl;
        }
}
```



Print_array gets the array pointer as a parm

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you
        cin  >> len;

        squares_array_ptr = new array with square
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```

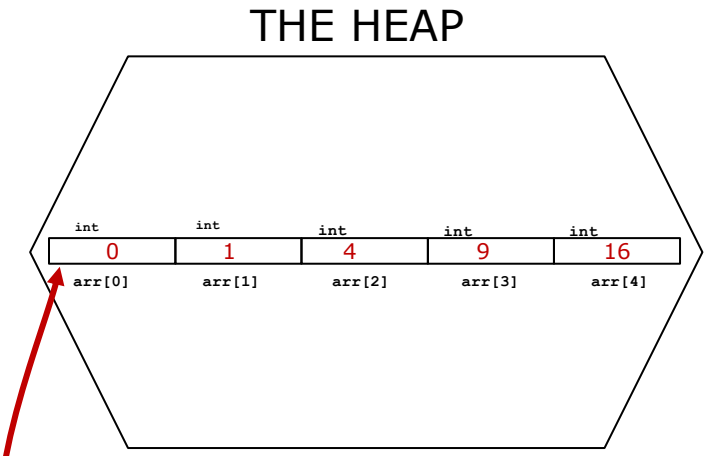
```
void print_array(int *arr_ptr, int length
{
        int i;

        for (i = 0; i < length; i++) {
                cout << "array[" << i << "] =  "
                     << arr_ptr[i] << endl;
        }
}
```



Note that although arr_ptr is typed as a pointer, print_array subscripts it.

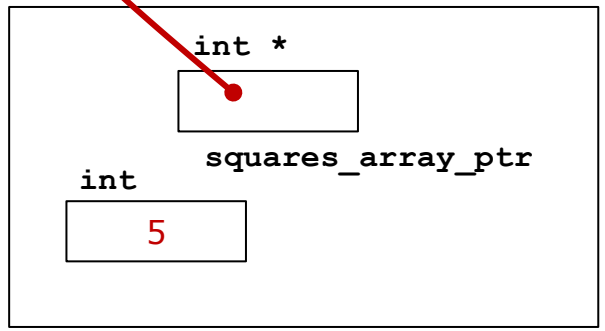
…and use the array until it's deleted!

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```



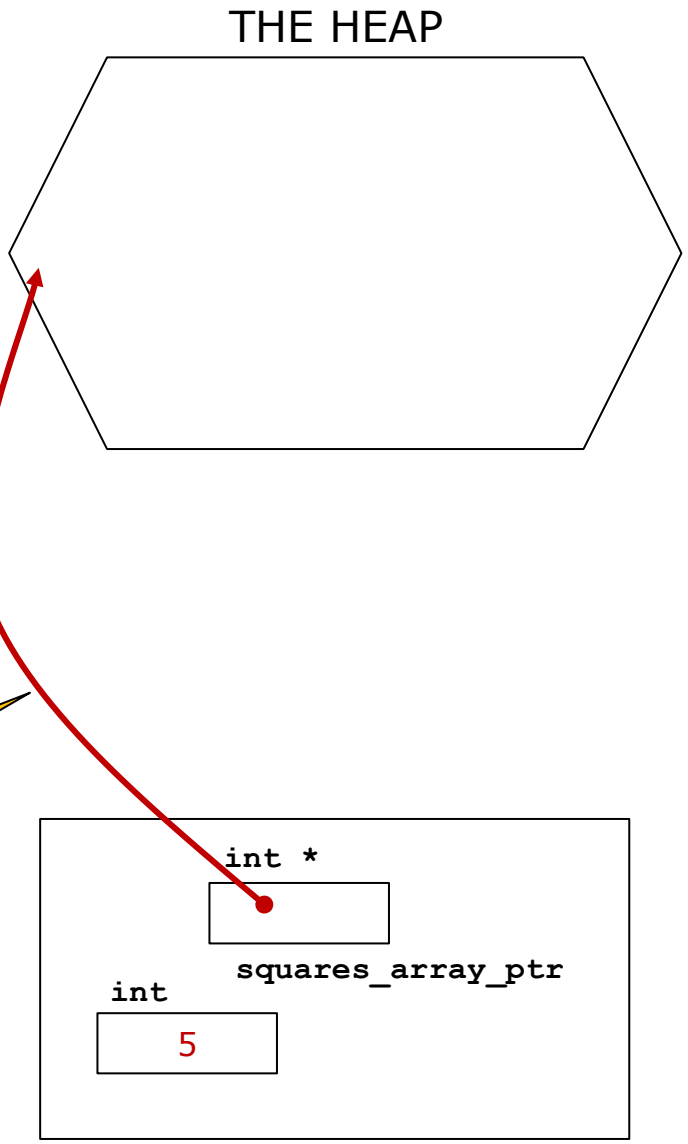
How many squares do you want to store? 5

```
array[0] =  0
array[1] =  1
array[2] =  4
array[3] =  9
array[4] =  16
```



…and use the array until it's deleted!

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```



If we weren't about to leave anyway, we should set squares_array_ptr=nullptr

What if we don't know how big the array should be?

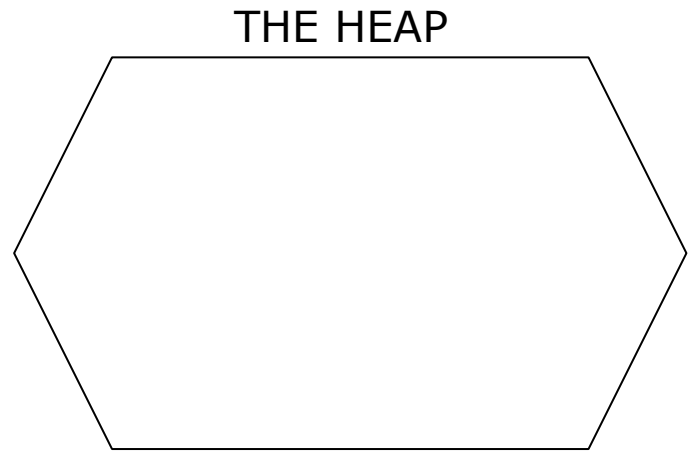
We'll build a dynamic list array!

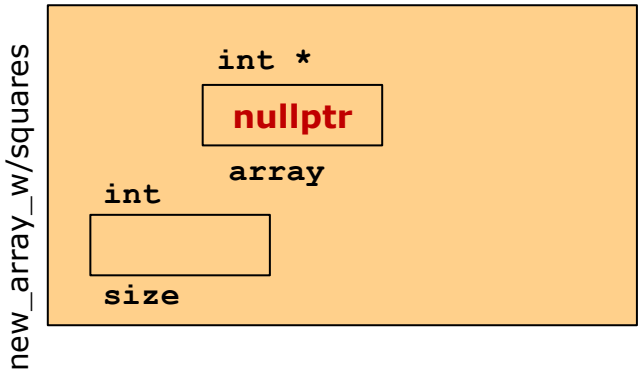
The idea…

- **We will build a service that can grow our arrays as necessary**
- **We can change our minds and ask for more each time, but…**
- **C++ arrays can't grow!**
- ***The function will reallocate a bigger array if necessary. If so it will:***
  - *Copy all the data from the old to the new*
  - *Always return the pointer to the latest array…the caller uses that!*

Calling a function that can grow arrays

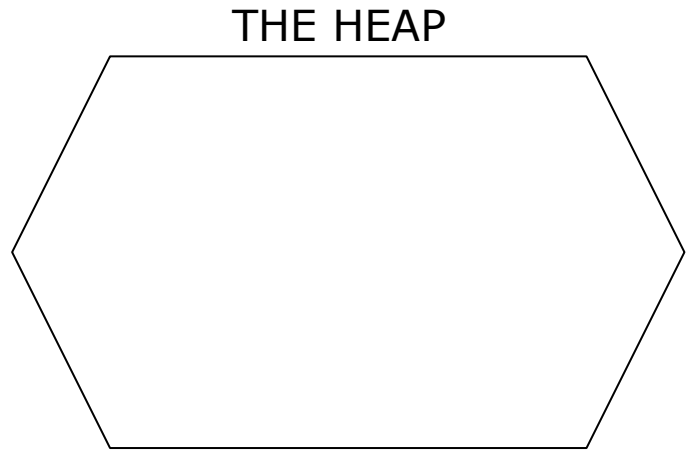
```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // Fill with squares
        for (int size = 0; size < target_size; size++) {
                // will bump array to size + 1
                //  but not change sizex
                array = grow_array_by_one(array, size);
                array[size] = size * size; }

        return array;
}
```

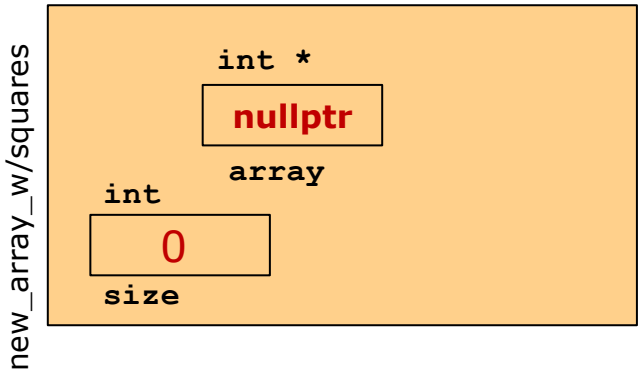




Calling a function that can grow arrays

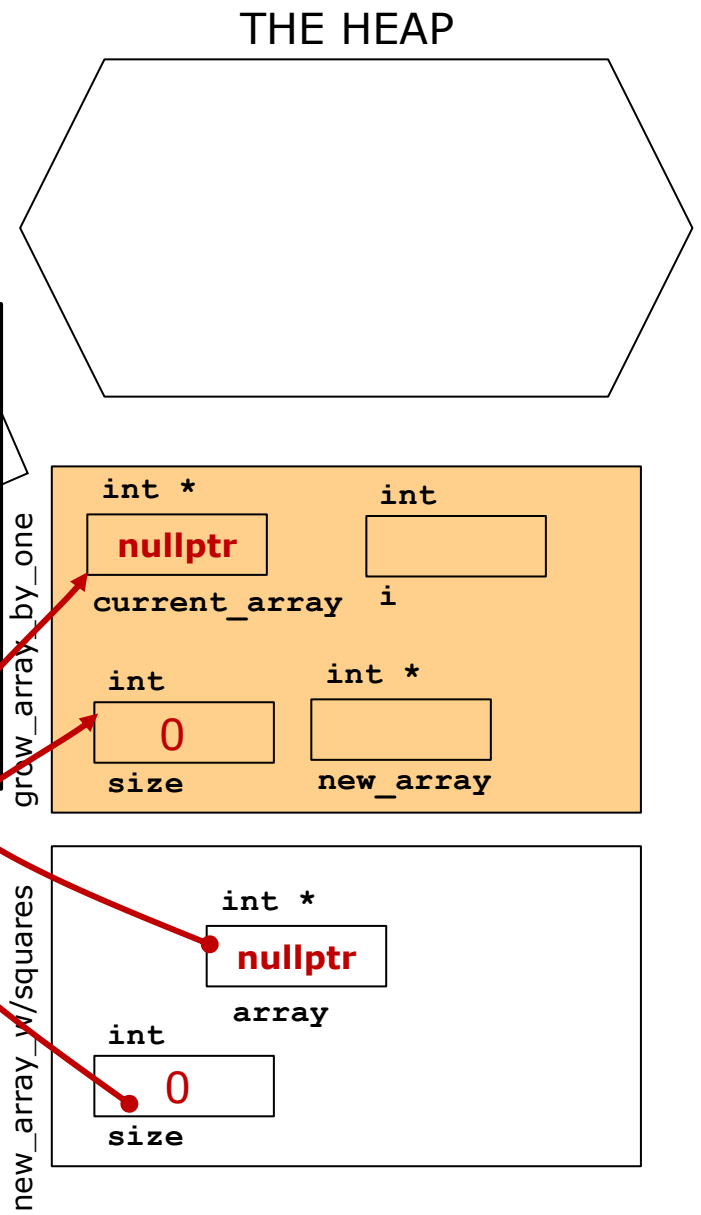
```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // Fill with squares
        for (int size = 0; size < target_size; size++) {
                // will bump array to size + 1
                //  but not change sizex
                array = grow_array_by_one(array, size);
                array[size] = size * size; }

        return array;
}
```





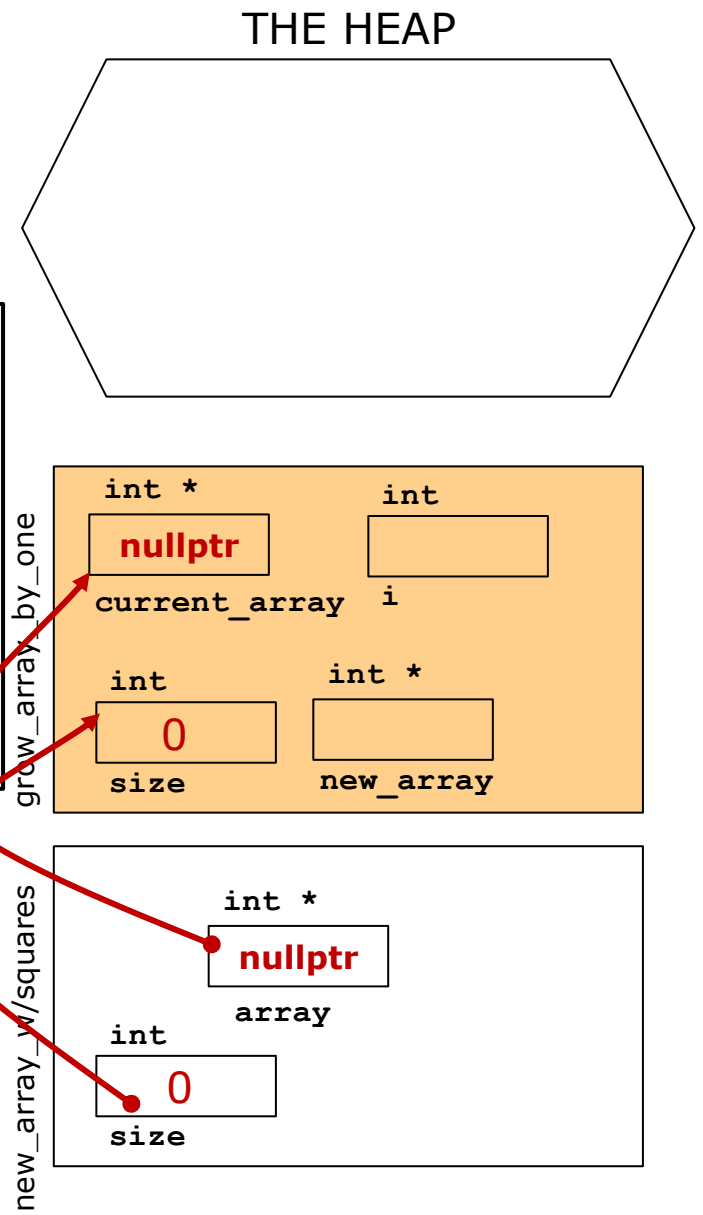
Allocating the first version of the array

```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;     // OK to use delete on nullptr
                                 // the first time through
    return new_array;
}
```



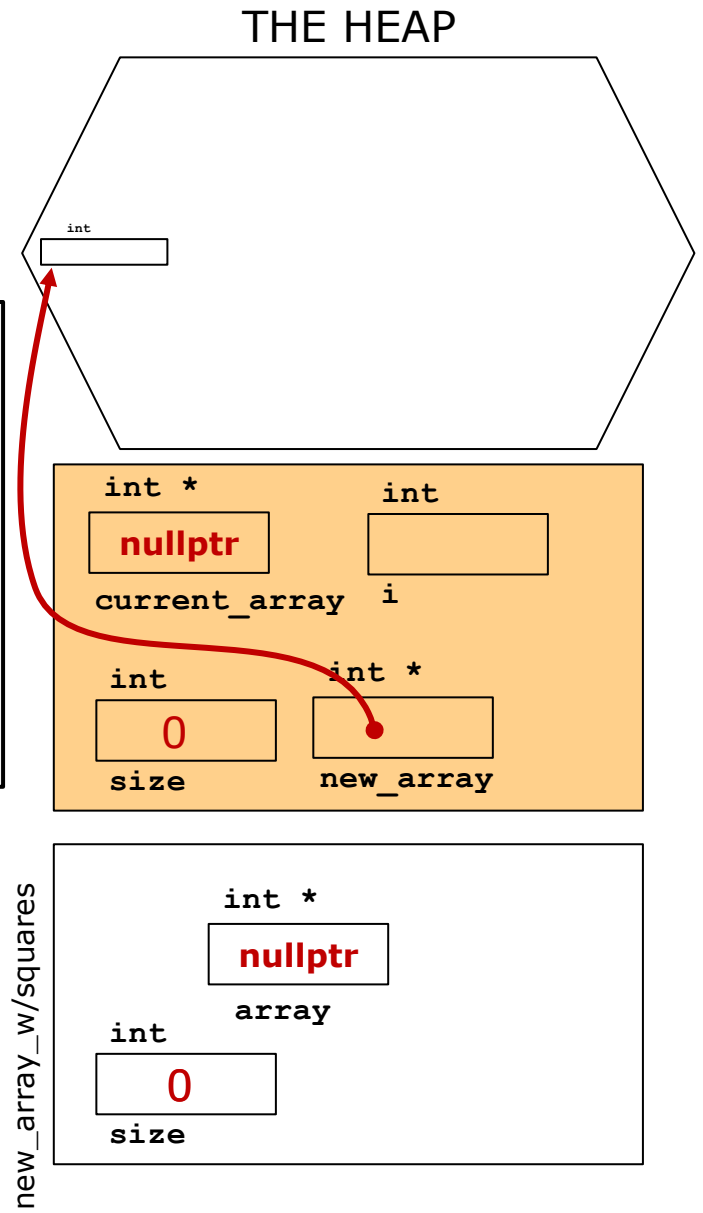
Allocating the first version of the array

```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;     // OK to use delete on nullptr
                                 // the first time through
    return new_array;
}
```



Allocating the first version of the array

```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;     // OK to use delete on nullptr
                                 // the first time through
    return new_array;
}
```

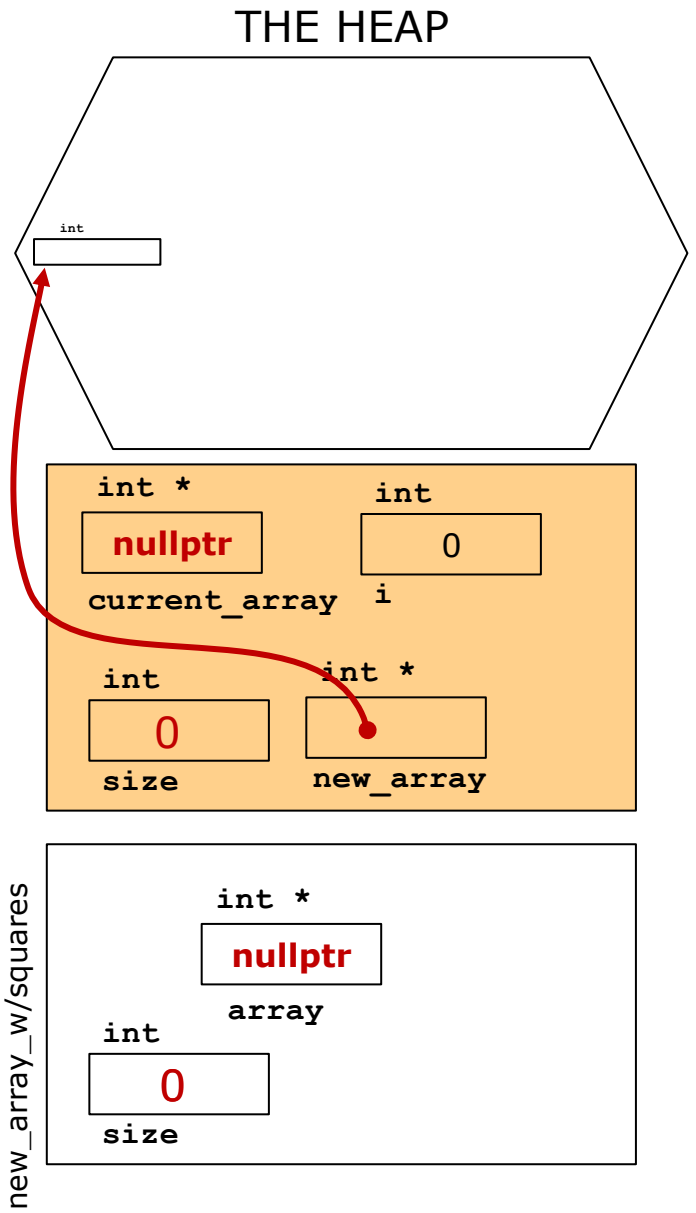


Allocating the first version of the array

```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;     // OK to use delete on nullptr
                                 // the first time through

    return new_ar
}
```

Does nothing this time

(size is 0)

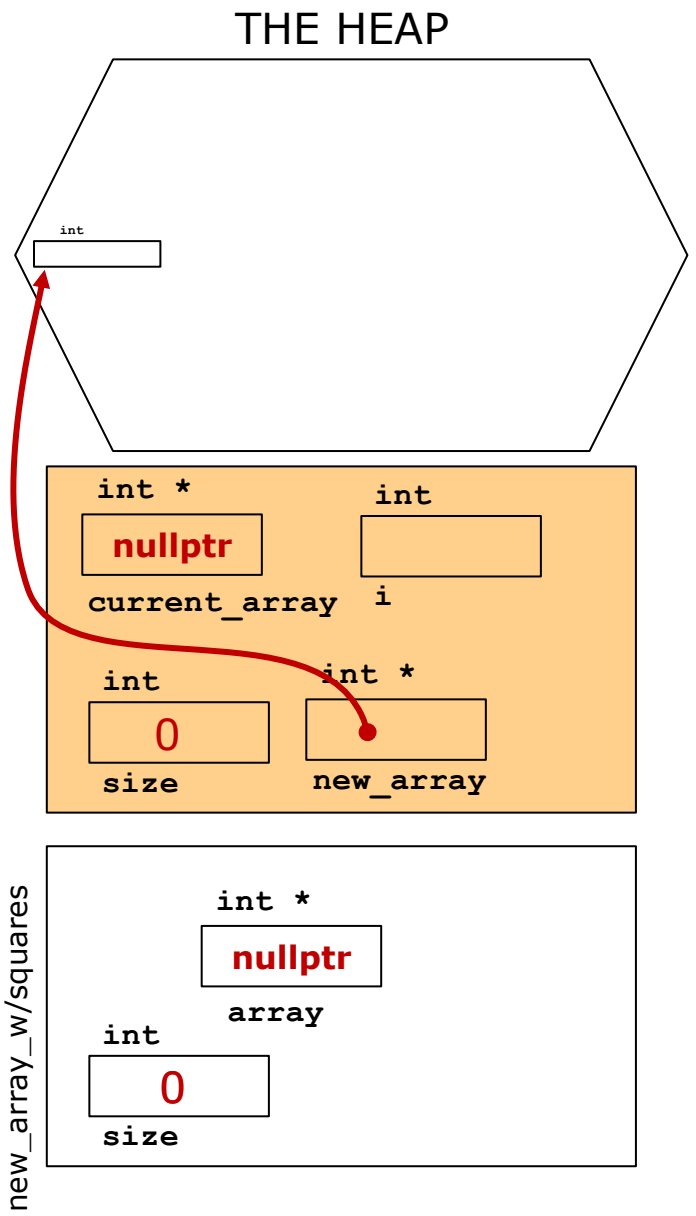


Allocating the first version of the array

```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;     // OK to use delete on nullptr
                                 // the first time through
    return new_array;
}
```

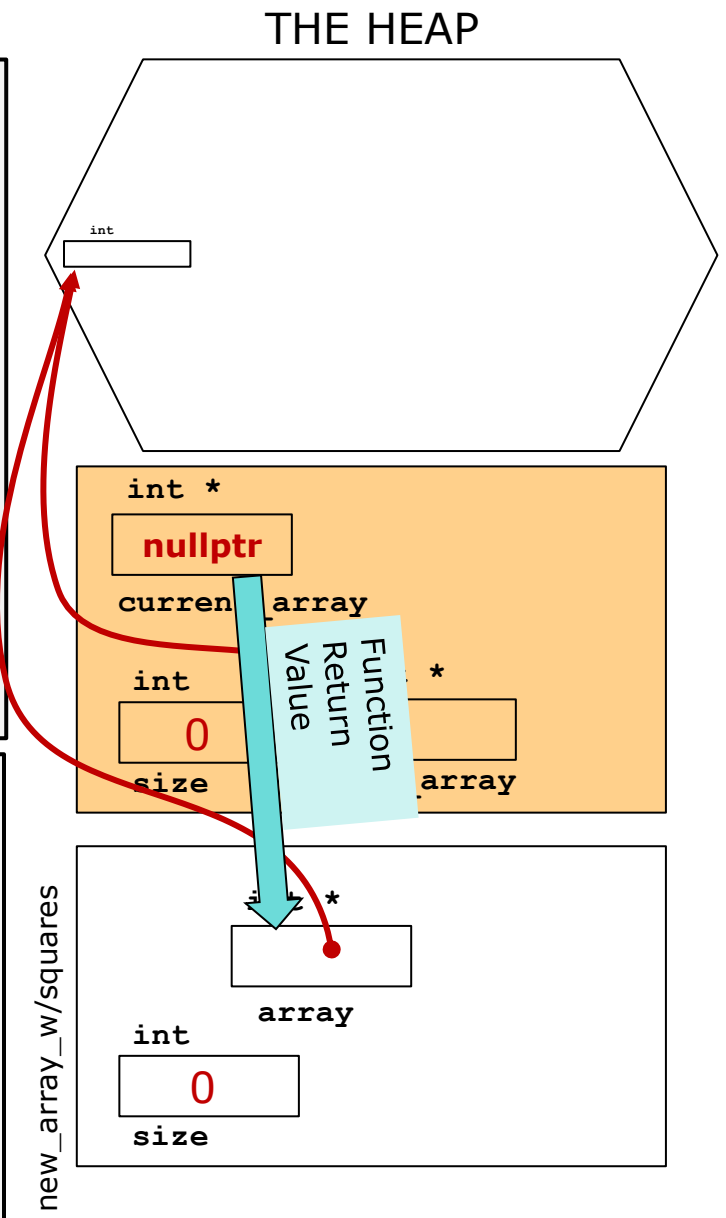
Does nothing this time

(`current_array` is `nullptr`)

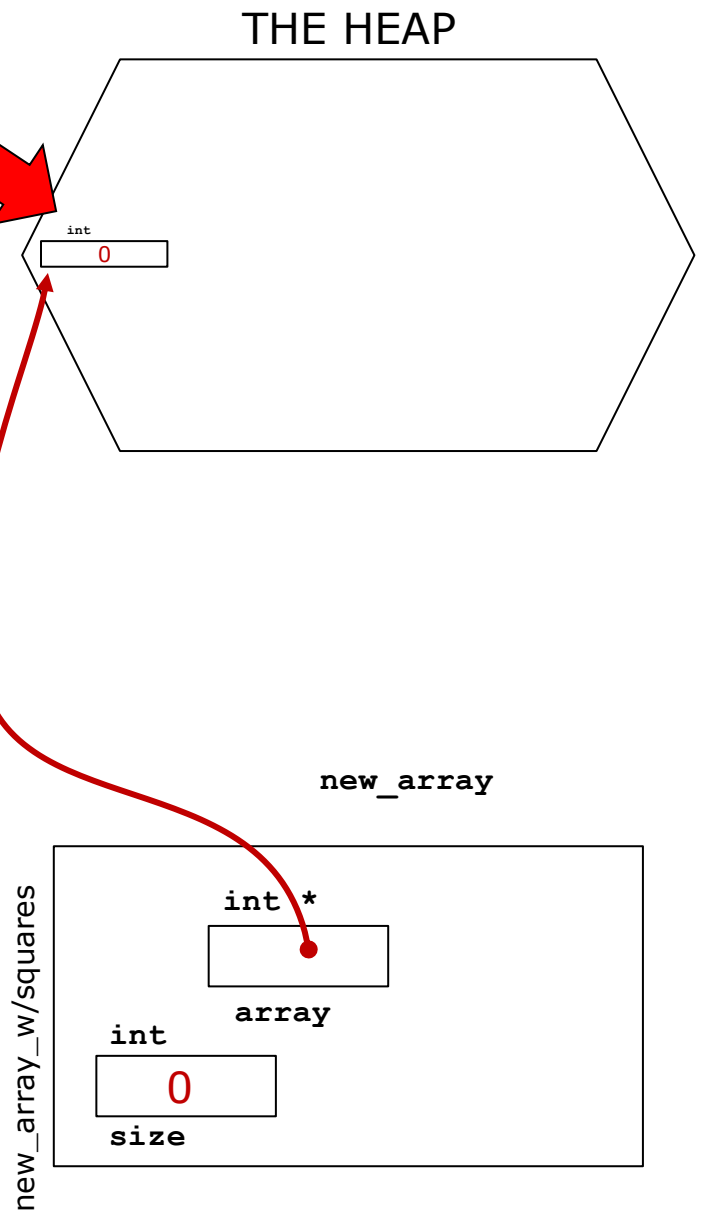


```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // Fill with squares
        for (int size = 0; size < target_size; size++) {
                // will bump array to size + 1
                //  but not change sizex
                array = grow_array_by_one(array, size);
                array[size] = size * size; }

        return array;
}
```

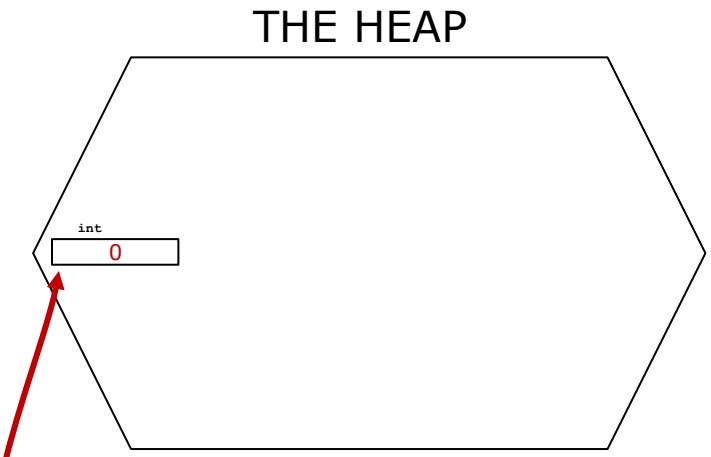
```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;     // OK to use delete on nullptr
                                 // the first time through
    return new_array;
}
```



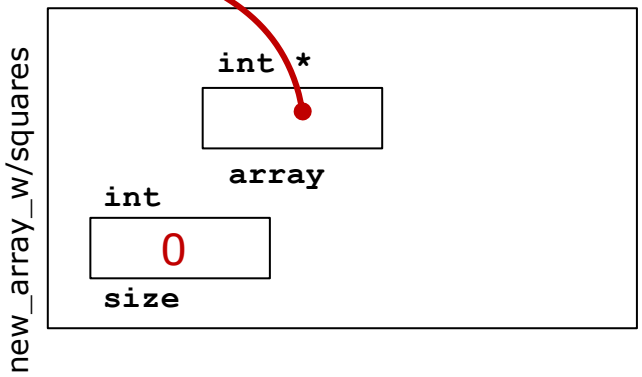
```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // Fill with squares
        for (int size = 0; size < target_size; size++) {
                // will bump array to size + 1
                //  but not change sizex
                array = grow_array_by_one(array, size);
                array[size] = size * size; }

        return array;
}
```



```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // Fill with squares
        for (int size = 0; size < target_size; size++) {
                // will bump array to size + 1
                //  but not change sizex
                array = grow_array_by_one(array, size);
                array[size] = size * size; }

        return array;
}
```



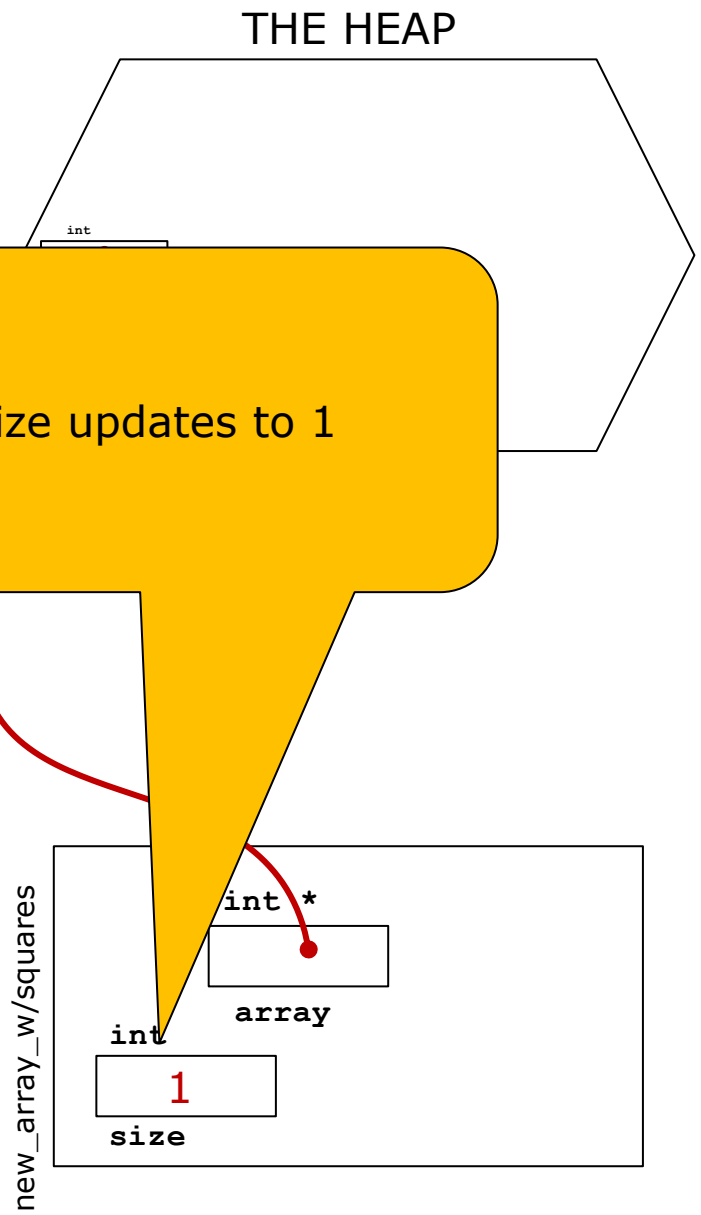
Now let's look at the second time around.



```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // Fill with squares
        for (int size = 0; size < target_size;
                // will bump array to size + 1
                //  but not change sizex
                array = grow_array_by_one(array,
                array[size] = size * size; }

        return array;
}
```

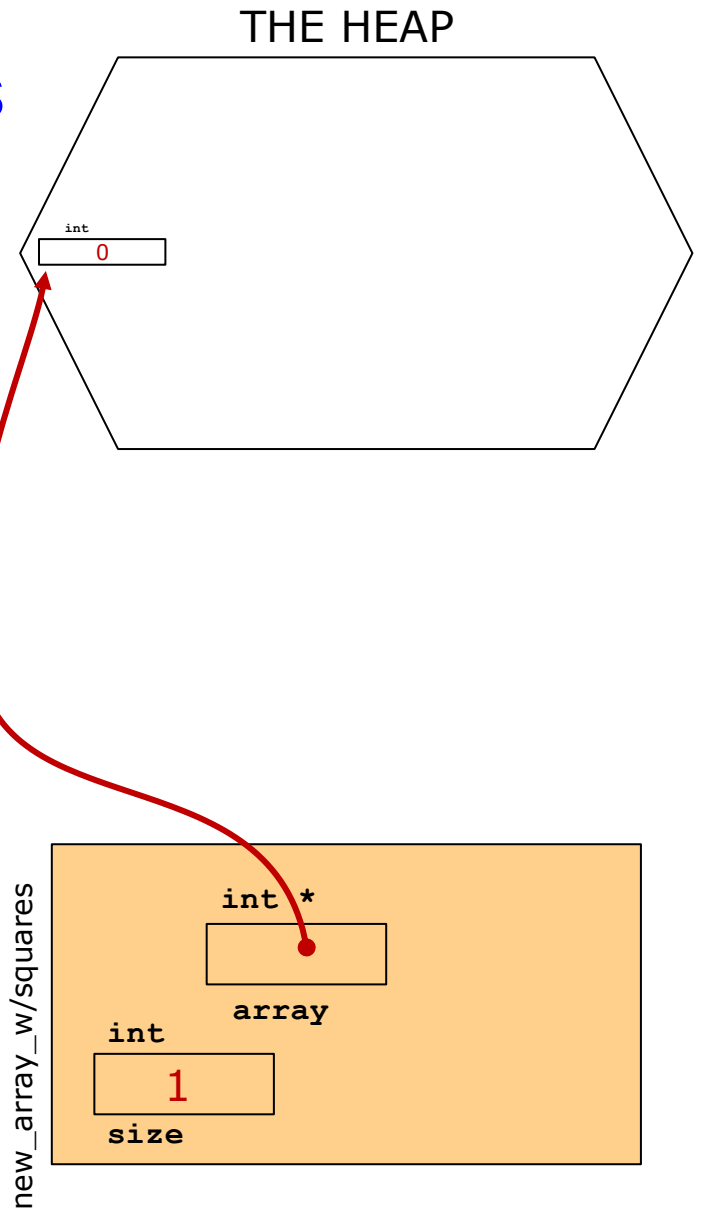
Caller's size updates to 1

Now let's look at the second time around.



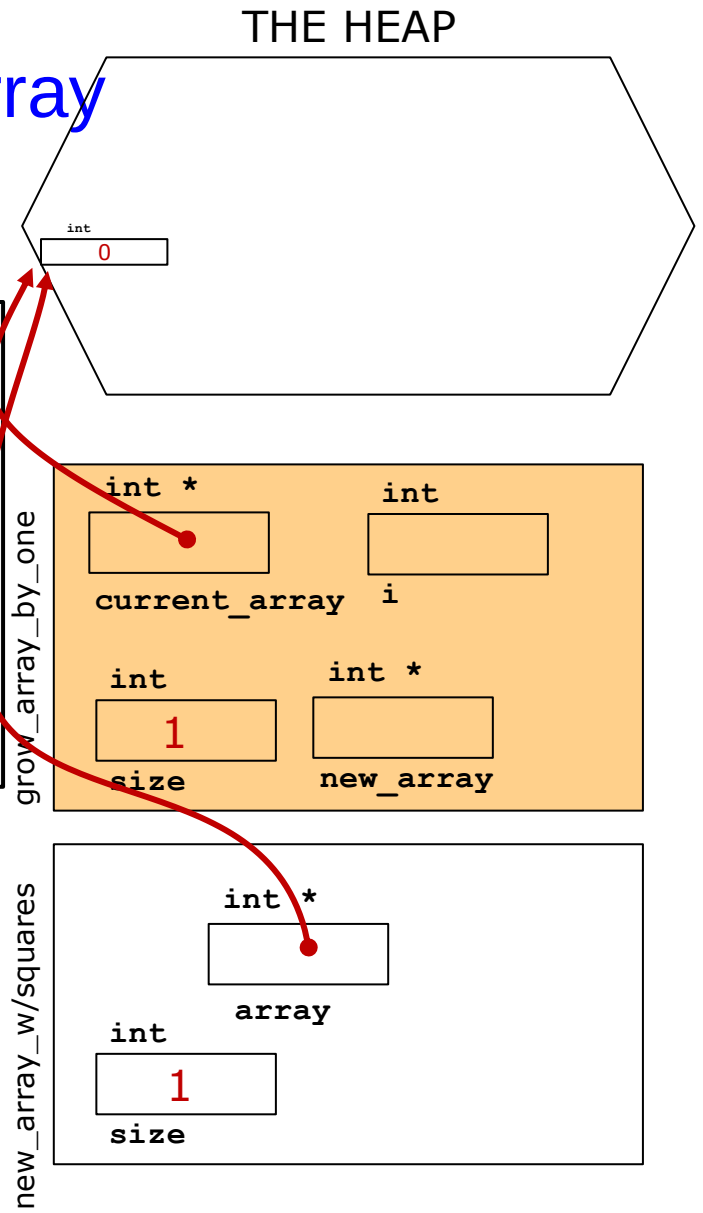
Calling a function that can grow arrays

```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // Fill with squares
        for (int size = 0; size < target_size; size++) {
                // will bump array to size + 1
                //  but not change sizex
                array = grow_array_by_one(array, size);
                array[size] = size * size; }

        return array;
}
```



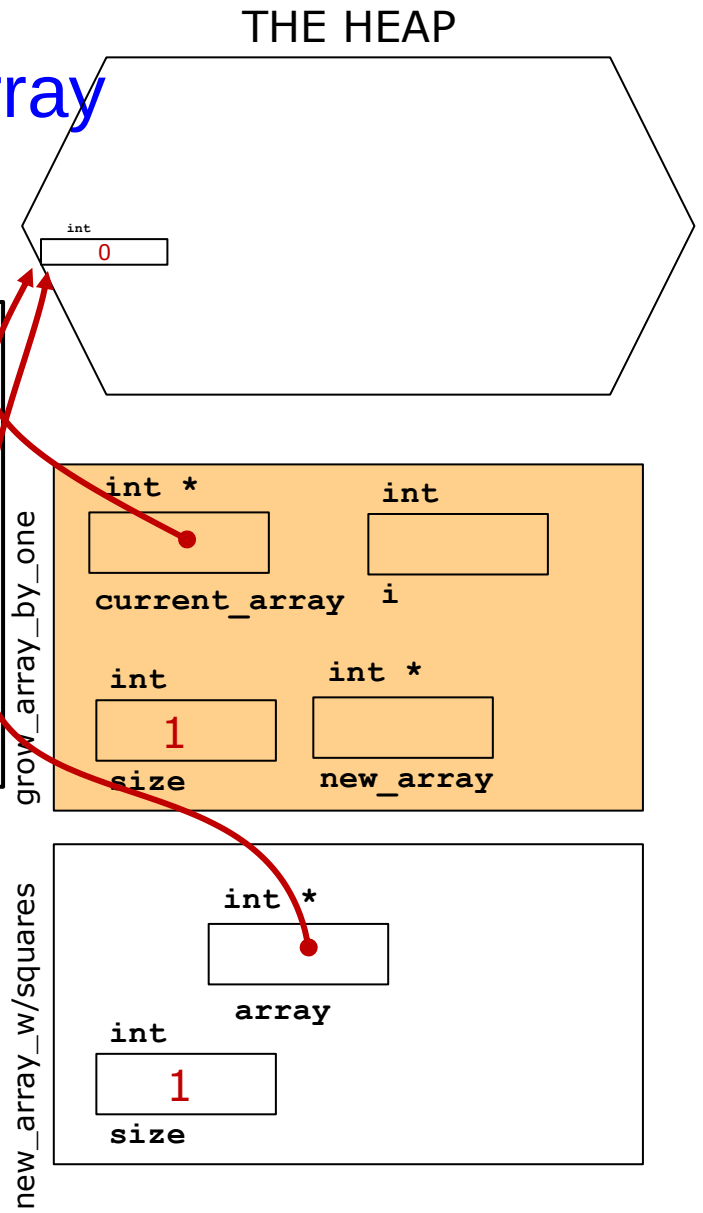
Allocating the *second* version of the array

```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;     // OK to use delete on nullptr
                                 // the first time through
    return new_array;
}
```



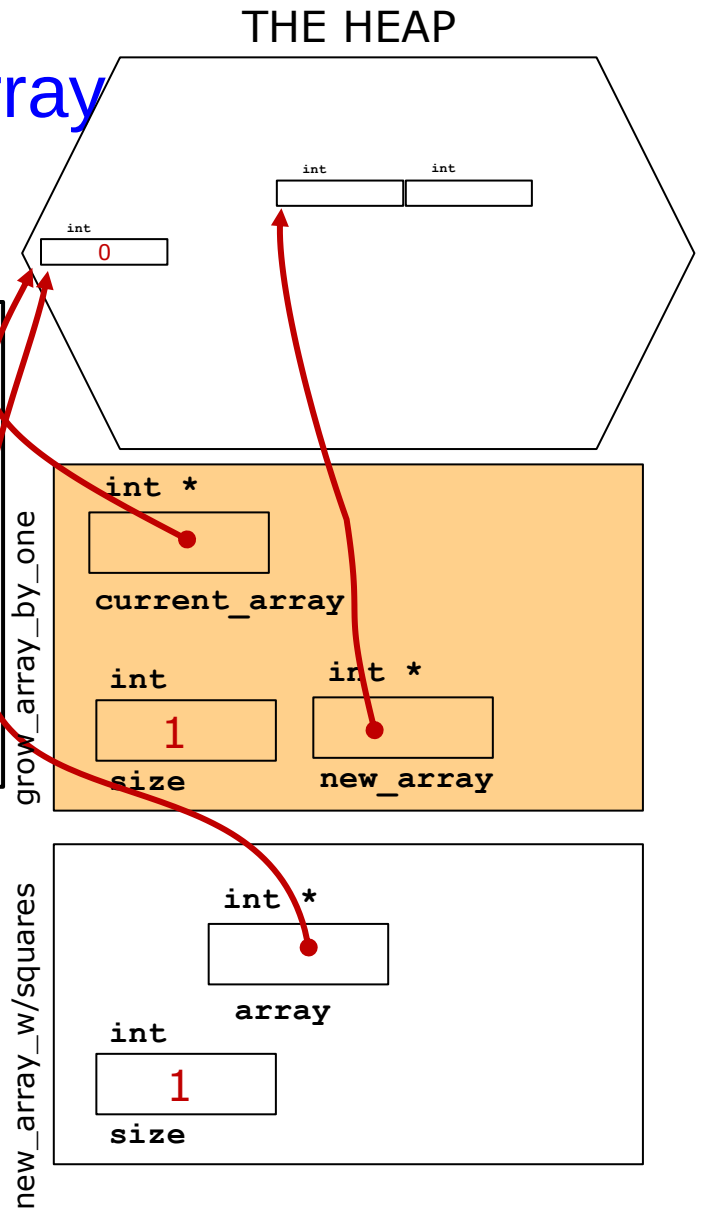
Allocating the *second* version of the array

```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;     // OK to use delete on nullptr
                                 // the first time through
    return new_array;
}
```



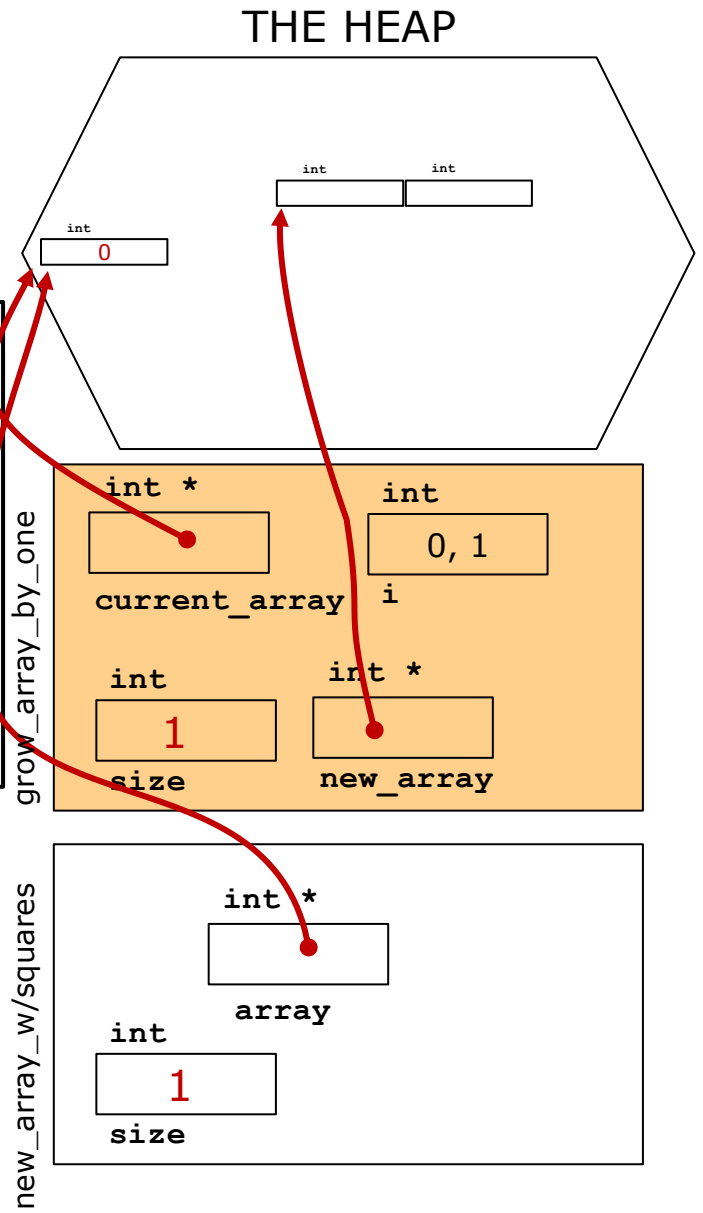
Allocating the *second* version of the array

```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;     // OK to use delete on nullptr
                                 // the first time through
    return new_array;
}
```



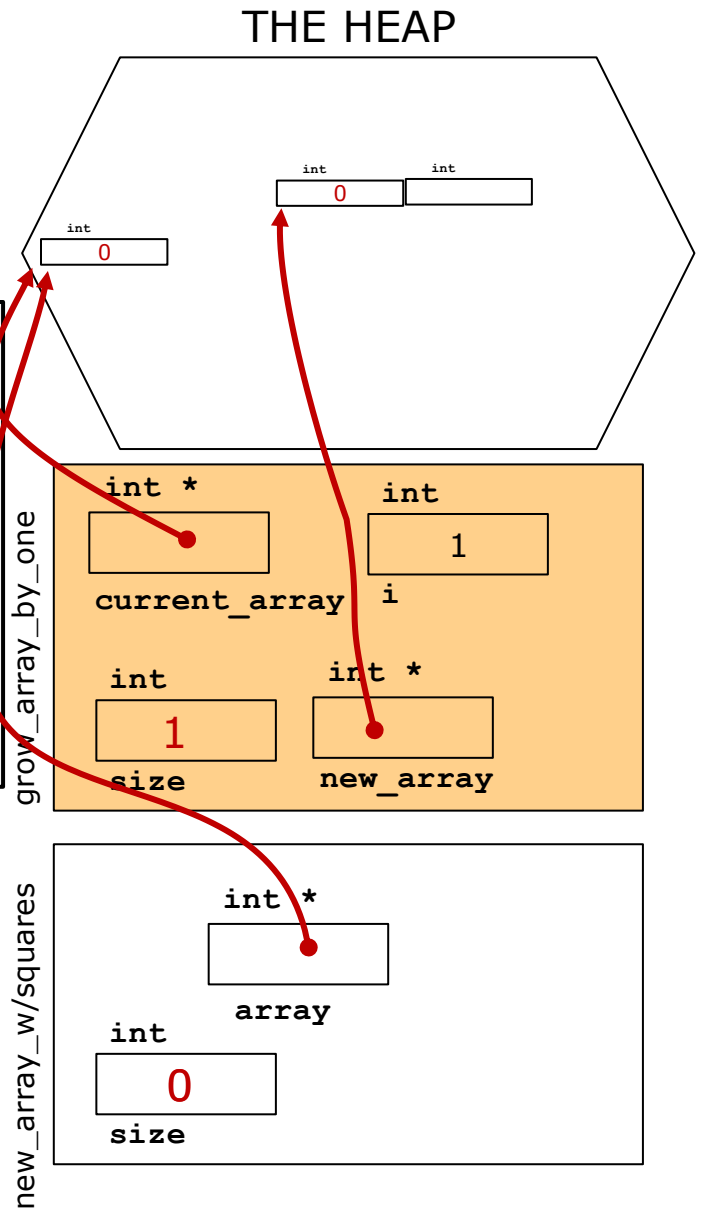
Copying the old to the new

```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;     // OK to use delete on nullptr
                                 // the first time through
    return new_array;
}
```



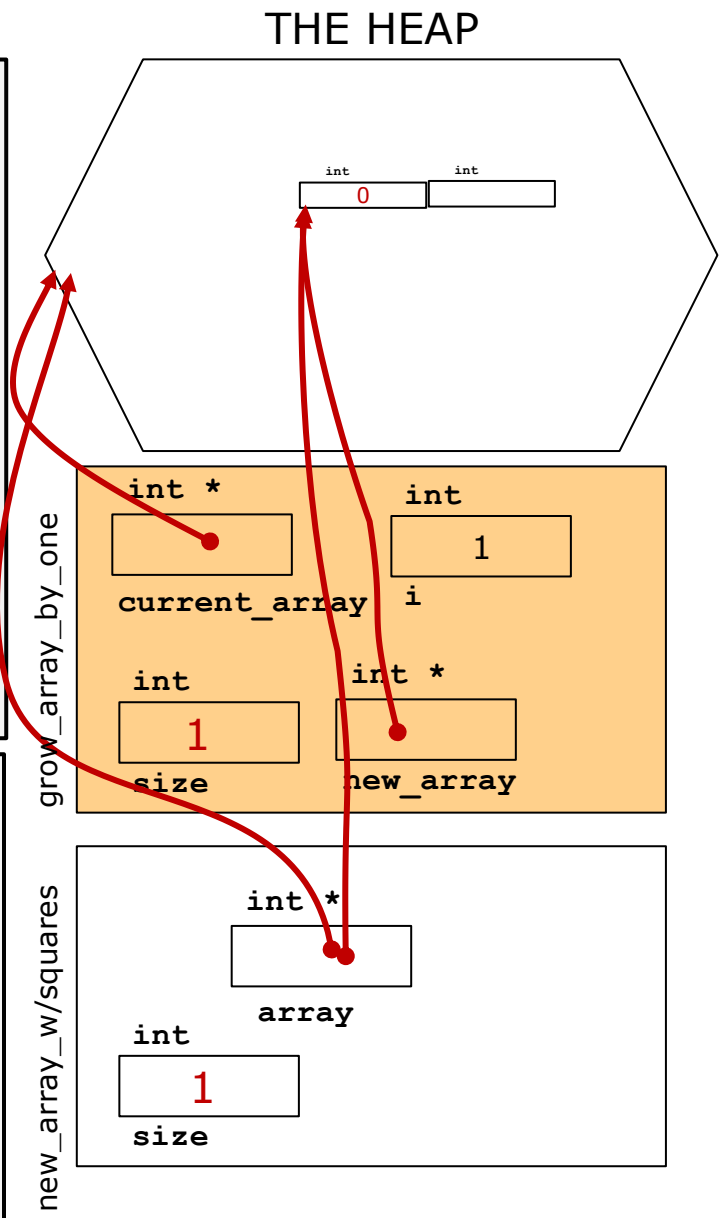
Deleting the one that's been replaced

```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;      // OK to use delete on nullptr
                                  // the first time through
    return new_array;
}
```

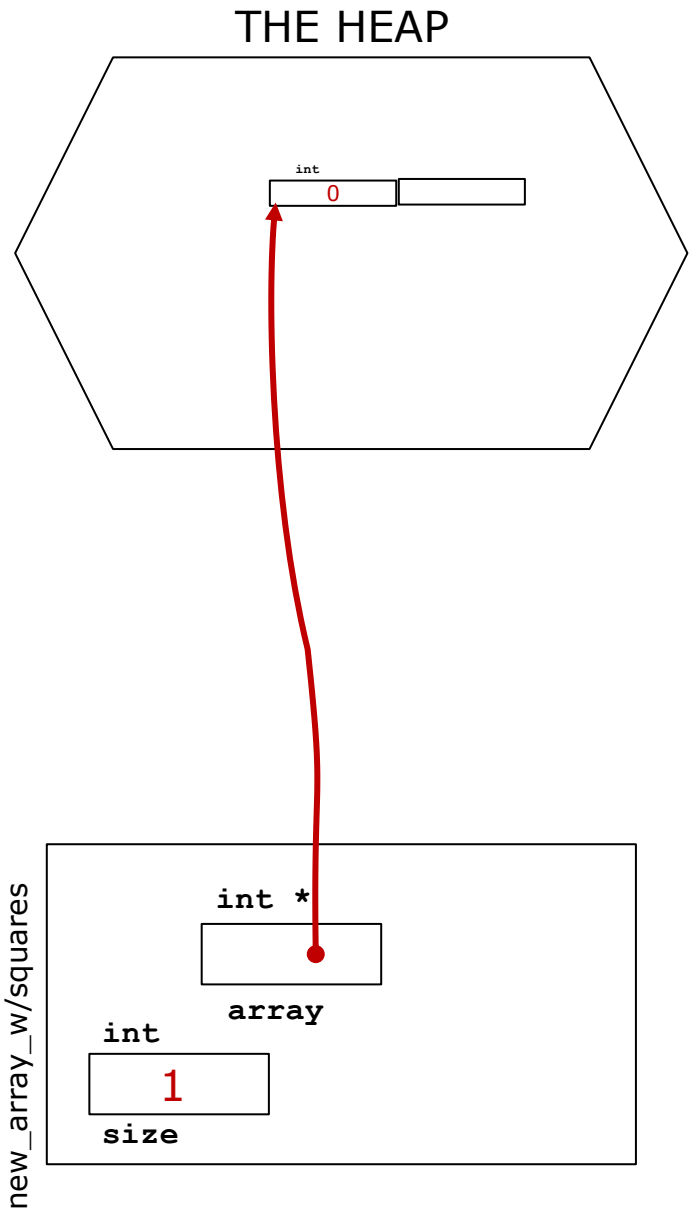


```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // Fill with squares
        for (int size = 0; size < target_size; size++) {
                // will bump array to size + 1
                //  but not change sizex
                array = grow_array_by_one(array, size);
                array[size] = size * size; }

        return array;
}
```

```
int *grow_array_by_one(int *current_array, int size)
{
    int i;
    int *new_array = new int [size + 1];

    for (i = 0; i < size; i++) {
        new_array[i] = current_array[i];
    }
    delete [] current_array;     // OK to use delete on nullptr
                                 // the first time through
    return new_array;
}
```



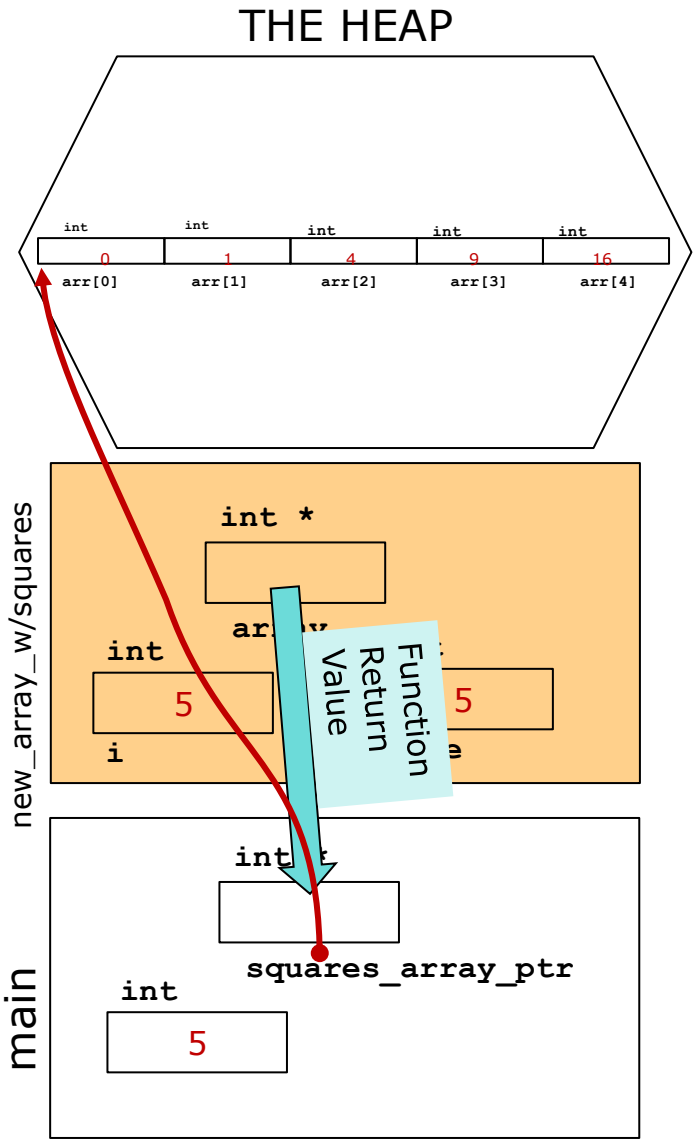
```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // Fill with squares
        for (int size = 0; size < target_size; size++) {
                // will bump array to size + 1
                //  but not change sizex
                array = grow_array_by_one(array, size);
                array[size] = size * size; }

        return array;
}
```



How does main get the array?

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```

```
int *new_array_with_squares(int size)
{
        int  i;
        int *array = new int [size];

        // An "identity array" sets each slot
        // value to the index of the slot
        for (i = 0; i < size; i++) {
                array[i] = i * i;   // array[i] gets i squared
        }
        return array;
}
```



Modularity!!

Can We Hide The Details
of the
Dynamic Array
from its users?

We can make this code much more modular

- **We'll put all the variables for each dynamic array into a structure**
- **Our squares application will *use functions* to perform all manipulations of the dynamic array.**
- ***These functions make the application cleaner and more modular***
- ***They abstract away (hide!) array details from the squares application***
- **We will sketch the details today…**
- **…on Tuesday this will lead us to a more robust approach using classes!**

The old squares program

```
int main ()
{
        int *squares_array_ptr;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array_ptr = new_array_with_squares(len);
        print_array(squares_array_ptr, len);
        delete [] squares_array_ptr;
}
```

The old squares program

Our new_array_with_squares_function had to manage the underlying array, and its length

```
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // Fill with squares
        for (int size = 0; size < target_size; size++) {
                // will bump array to size + 1 but not change size
                array = grow_array_by_one(array, size);
                array[size] = size * size;   // array[i] gets i squared
        }

        return array;
}
```

The new squares program

Details of the array implementation are in a Smart_Array struct.

```
int main ()
{
        Smart_Array squares_array;
        int  len;

        cout << "How many squares do you want to store? ";
        cin  >> len;

        squares_array = new_array_with_squares(len);
        print_array(squares_array);
        delete_smart_array(&squares_array);
}
```

The new_array_with_squares func

Smart set_at function is like:

arr[i] = …

…but *automatically grows the array as needed*.

```
Smart_Array new_array_with_squares(int targe
{

        Smart_Array arr = new_smart_ar      ();

        // Fill with squares
        for (int i = 0; i < tar_et_size; i++) {
                // This will grow the array as needed
                set_at(&arr, i, i*i);   // array[i] gets i squared
        }

        return arr;
}
```

The Smart_Array Struct

This struct encapsulates both the pointer to the current allocated array, and its size.

```
struct Smart_Array {
        int *current_array;
        int size;   // highest set index+1
};
```

Example of two of the Smart array functions

```
struct Smart_Array {
        int *current_array;
        int size;   // highest set index+1
};
```

```
int get_at(Smart_Array arr, int i)
{
        // Here we assume that all requests are
        // properly within the size of the array

        return arr.current_array[i];
}
```

Example of two of the Smart array

This struct encapsulates both the pointer to the current allocated array, and its size.

```
struct Smart_Array {
        int *current_array;
        int size;   // highest [illegible] index+1
};
```

```
void set_at(Smart_Array *arr_ptr, int i, int new_value)
{
        while(arr_ptr->size <= i) {
                grow_array(arr_ptr);
        }
        arr_ptr->current_array[i] = new_value;
}
```

Example of two of the Smart array

In the end we'll update the array in the obvious way, but…

```
struct Smart_Array {
        int *current_array;
        int size;   // highest se          x+1
};
```

```
void set_at(Smart_Array *arr_          , int i, int new_value)
{
        while(arr_ptr->size          = i) {
                grow_array(          rr_ptr);
        }
        arr_ptr->current_array[i] = new_value;
}
```

Example of two of the Smart array functions

```
struct Smart_Array {
        int *current_array;
        int size;   // highest set      +1
};
```

We automatically grow the array for the caller *if* it's not already big enough.

```
void set_at(Smart_Array *arr_ptr, int i, int new_value)
{
        while(arr_ptr->size <= i) {
                grow_array(arr_ptr);
        }
        arr_ptr->current_array[i] = new_value;
}
```

Example of two of the Smart array functions

```
struct Smart_Array {
        int *current_array;
        int size;   // highest
};
```

Grow_array may update the struct by allocating a new array and changing the size, thus we must pass it into this function *by reference,*

I.e. we receive here the *address* of the caller's Smart_Array struct, not a copy.

```
void set_at(Smart_Array *arr_ptr, int i, int new_value)
{
        while(arr_ptr->size <= i) {
                grow_array(arr_ptr);
        }
        arr_ptr->current_array[i] = new_value;
}
```

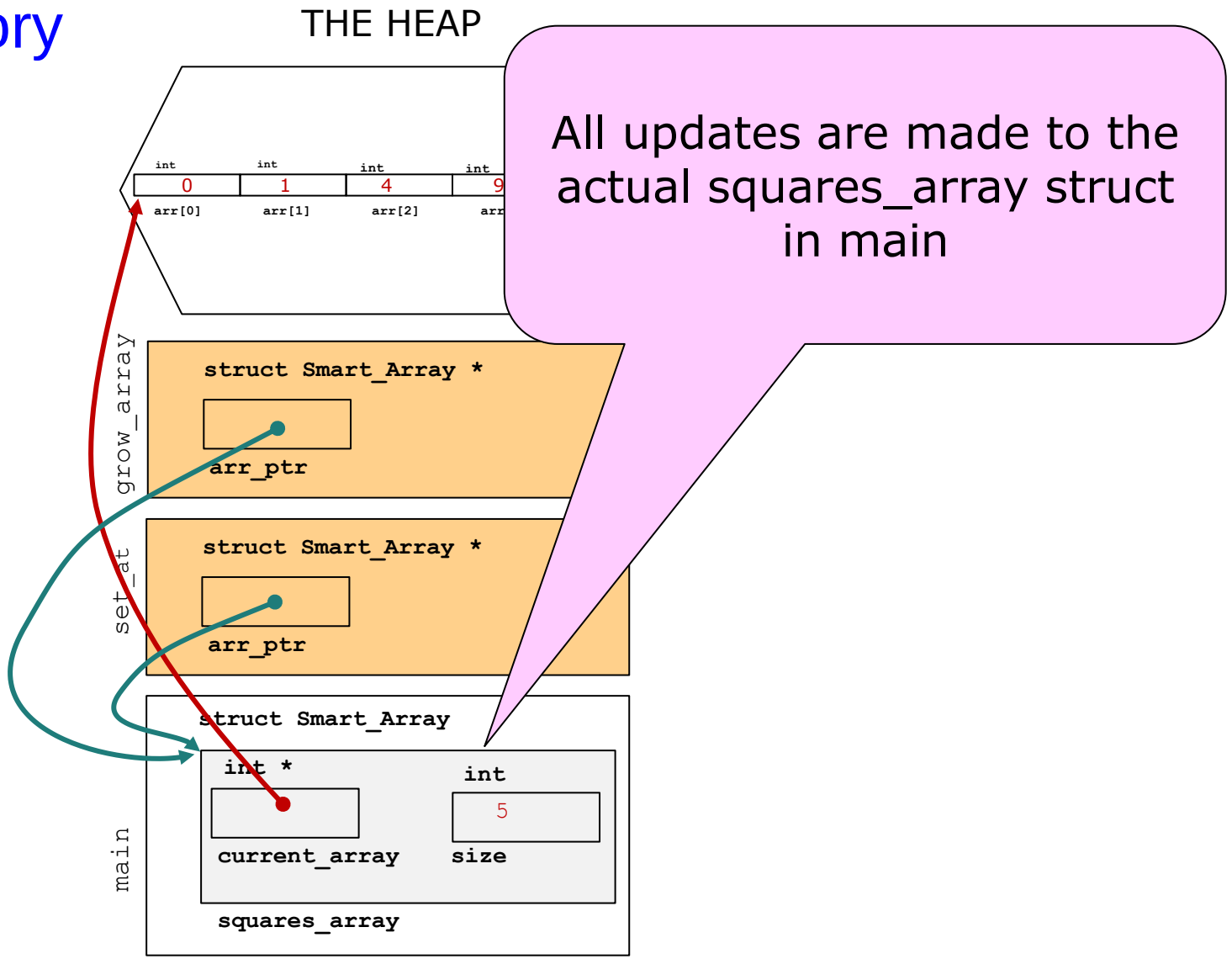
Example of two of the Smart array functions

```
struct Smart_Array {
        int *current_array;
        int size;   // highest
};
```

..and for the same reason we pass on the pointer to grow_array, which is actually manipulating the struct in main.

```
void set_at(Smart_Array *arr_ptr, int i, int new_value)
{
        while(arr_ptr->size <= i) {
                grow_array(arr_ptr);
        }
        arr_ptr->current_array[i] = new_value;
}
```

What's in memory



The `grow_array` function

```
struct Smart_Array {
        int *current_array;
        int size;   // highes
};
```

Using the arr_ptr argument, grow_array is updating the struct that's on the stack in *main's activation record.*

```
void grow_array(Smart_Array *arr_ptr)
{
        int i;
        int *new_array = new int[arr_ptr->size + 1];
        for (i = 0; i < arr_ptr->size; i++) {
                new_array[i] = arr_ptr->current_array[i];
        }

        delete [] arr_ptr->current_array;
        arr_ptr->size = arr_ptr->size + 1;
        arr_ptr->current_array = new_array;
}
```

Review: Using structs and functions for modularit

- **We've seen how the application code got much simpler**
- **Details of managing the arrays were hidden…**
- **…e.g. we can change the array growing strategy without changing the code that makes and prints lists of squared numbers**
- **The pointer stuff looks a little tricky at first, but these are classic techniques…they're not hard once you get the hang of it!**
- **In our next lesson we will see how by turning our structs into classes, we can make the story even more powerful**
- **…a little later we will learn to package *classes* like Smart_Array into separate, shareable C++ sources files so we can reuse them in lots of programs!**

Summary