CS 11: Introduction to Computer Science

From Arrays to Lists

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

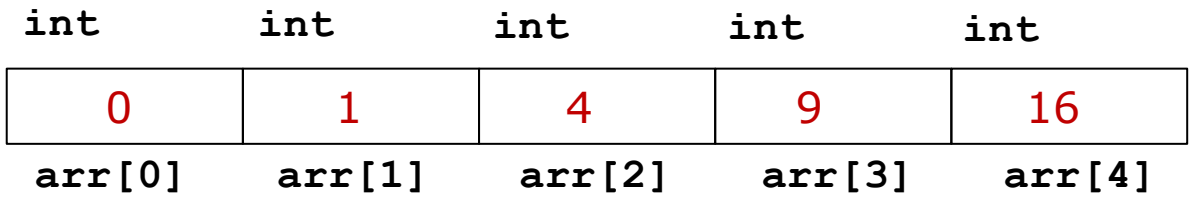
- **Lists: one of the most important CS data structures**
  - What "real world" needs do they meet?
  - Examples of lists
- **The same List *interface* can be *implemented* many ways**
- **The Array Lists we've been building are lists implemented as arrays**
- **First glimpse: linked lists**
- **But first: detailed review of SmartArrays**

Review: Building List *Arrays*

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

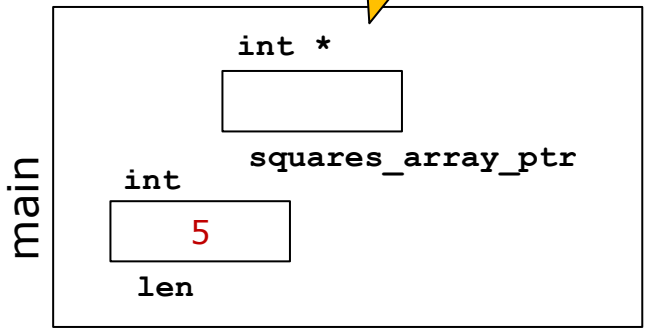


But where do the variables live?

```
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.

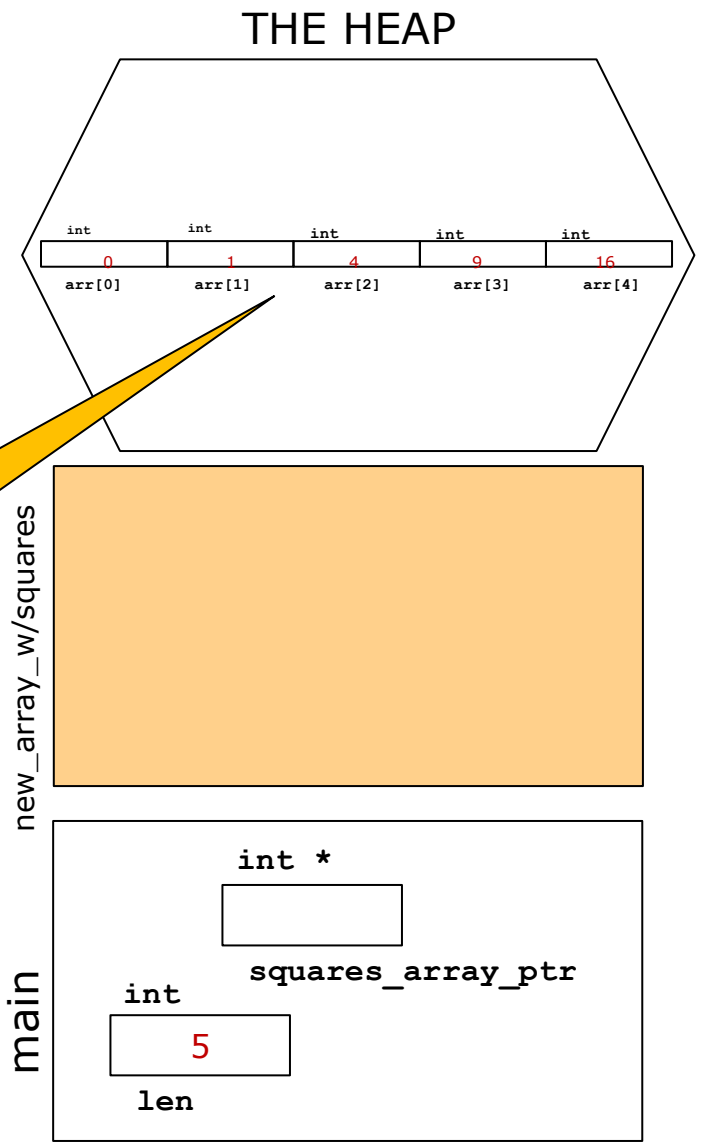


But where do the variables live?

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

5

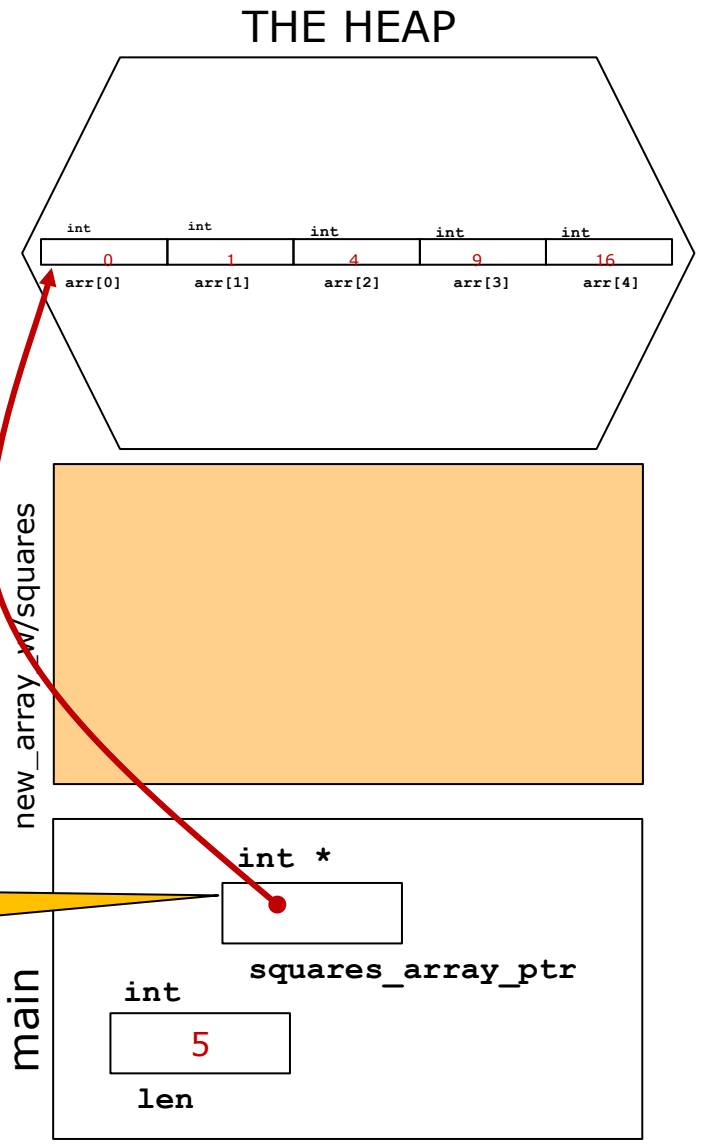
new_array_with_squares will allocate it on the heap using new…



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;
}
```

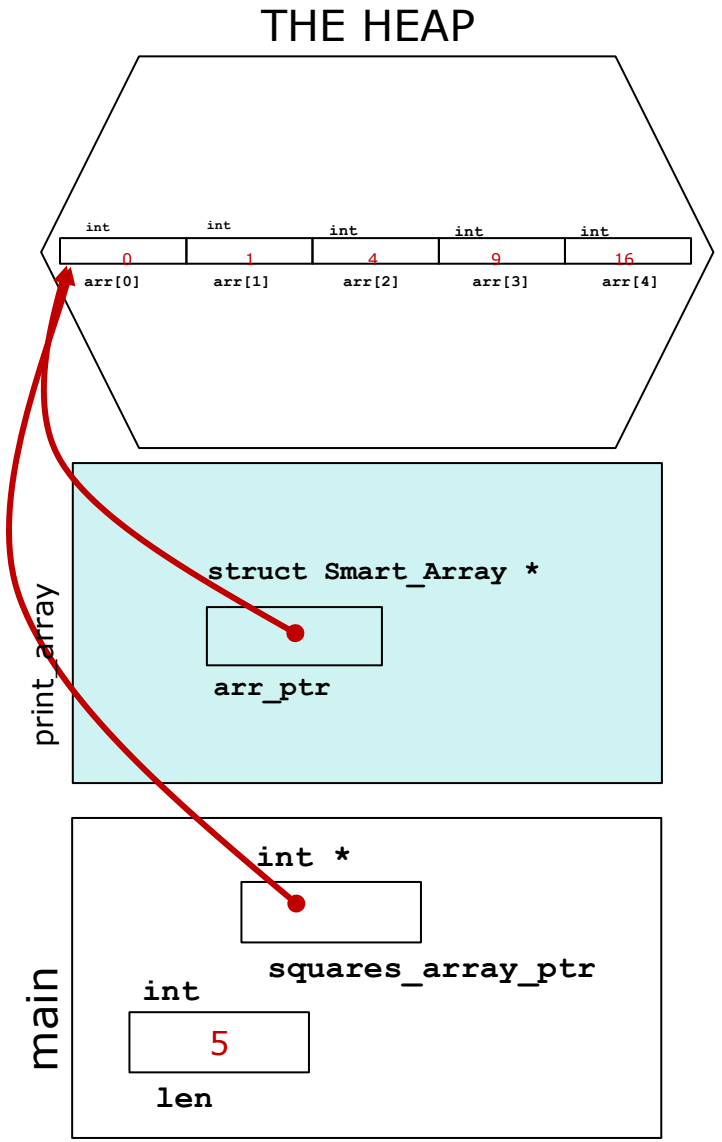
…and will return a pointer that we can keep here.



But where do the variables live?

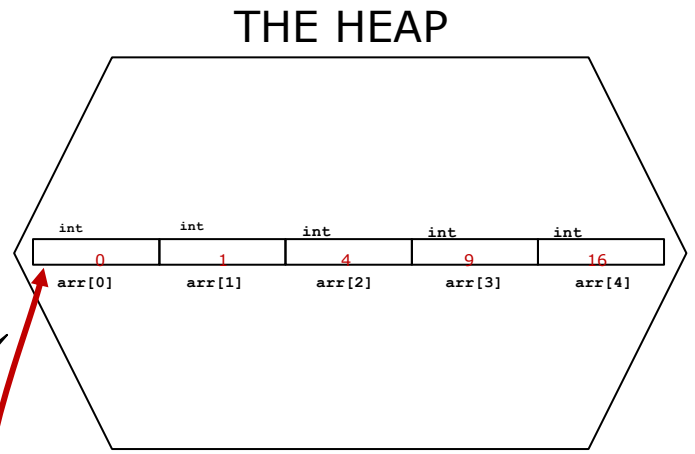
```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        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;
}
```

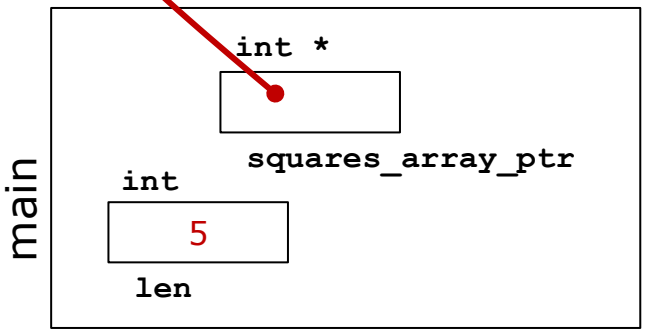
…so we can pass the pointer into the printing function which can then find the array!



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;
}
```

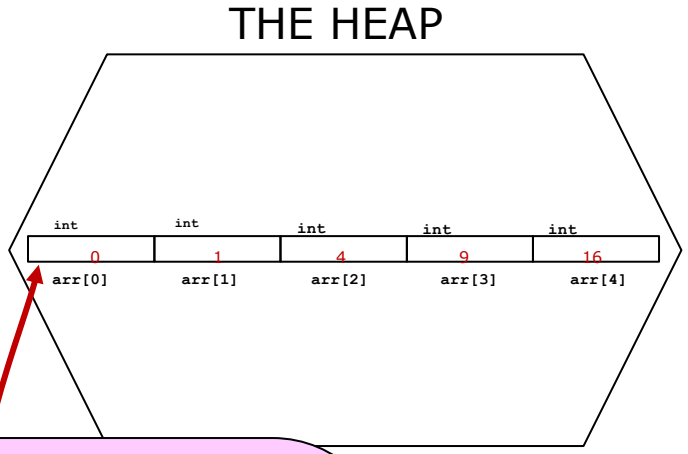




Crucially…

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?



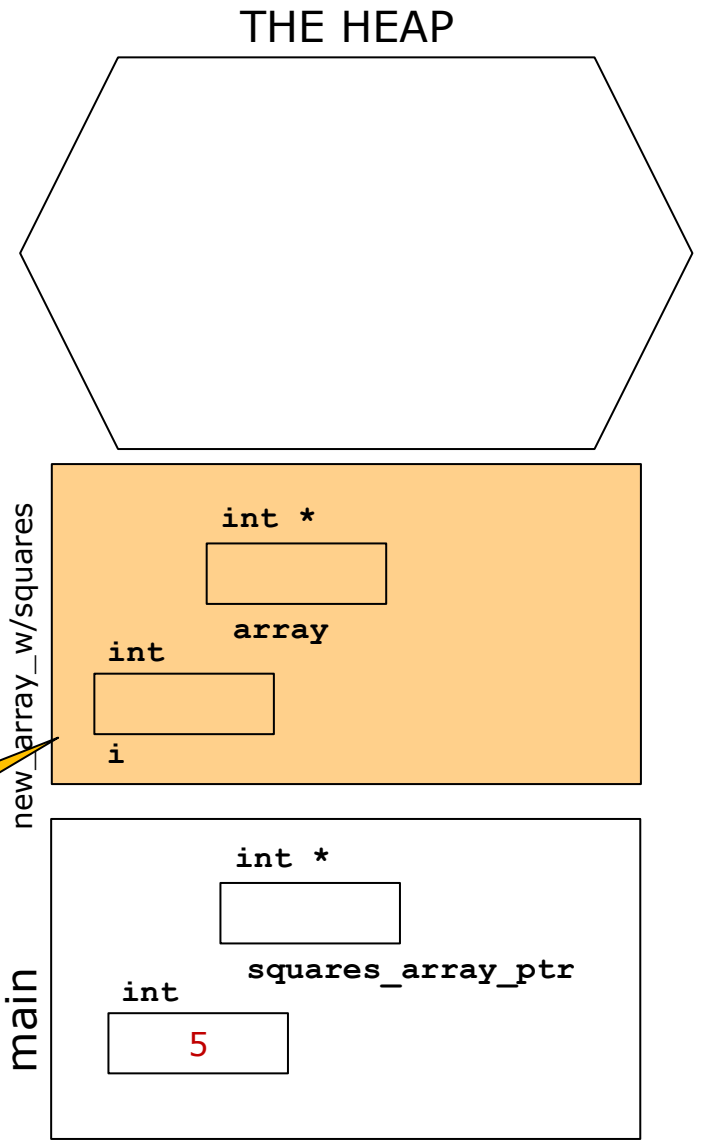

```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int
}
```

array_ptr

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

Creating the array on the heap

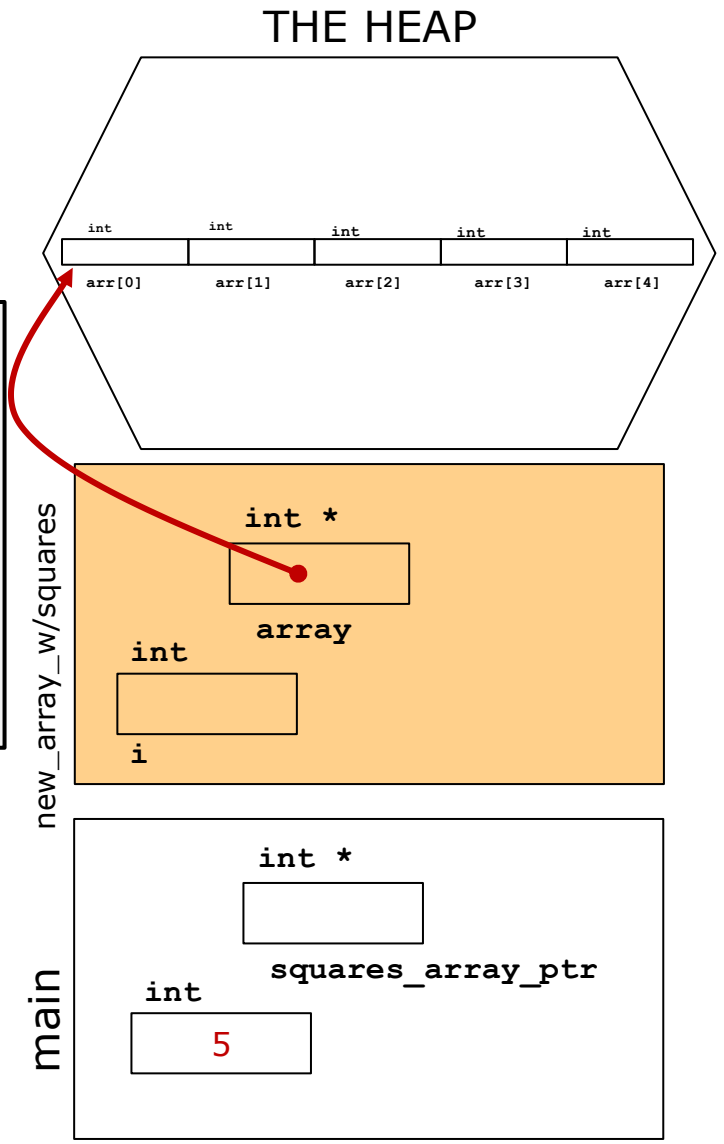
```
int *new_array_with_squares(int size)
{
        int  i;
        int *array = new int [size];

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



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

Creating the array on the heap

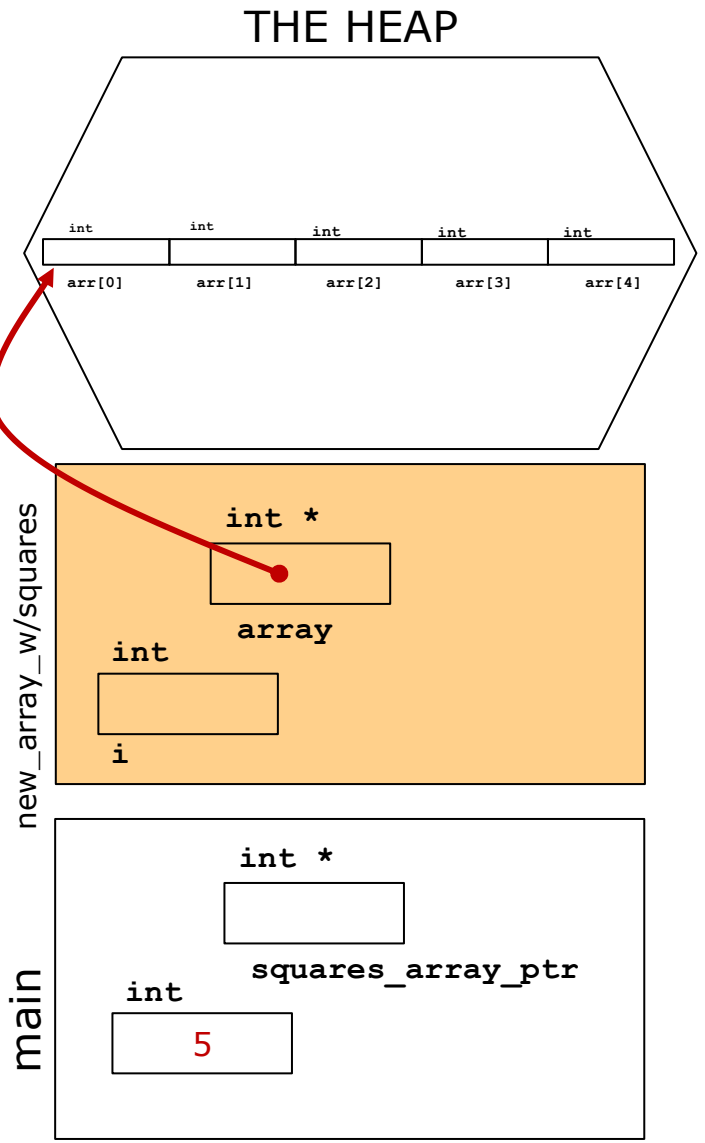
```
int *new_array_with_squares(int size)
{
        int  i;
        int *array = new int [size];

        // 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

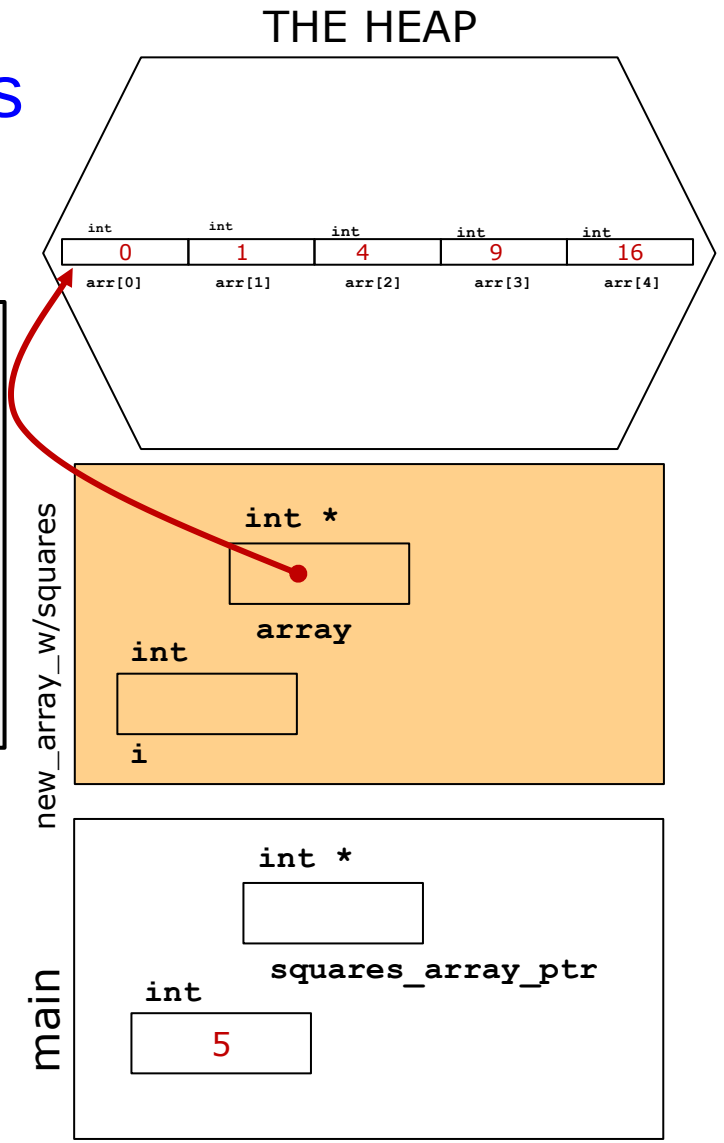
The C++ new operation allocates the array *on the heap!*

```
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

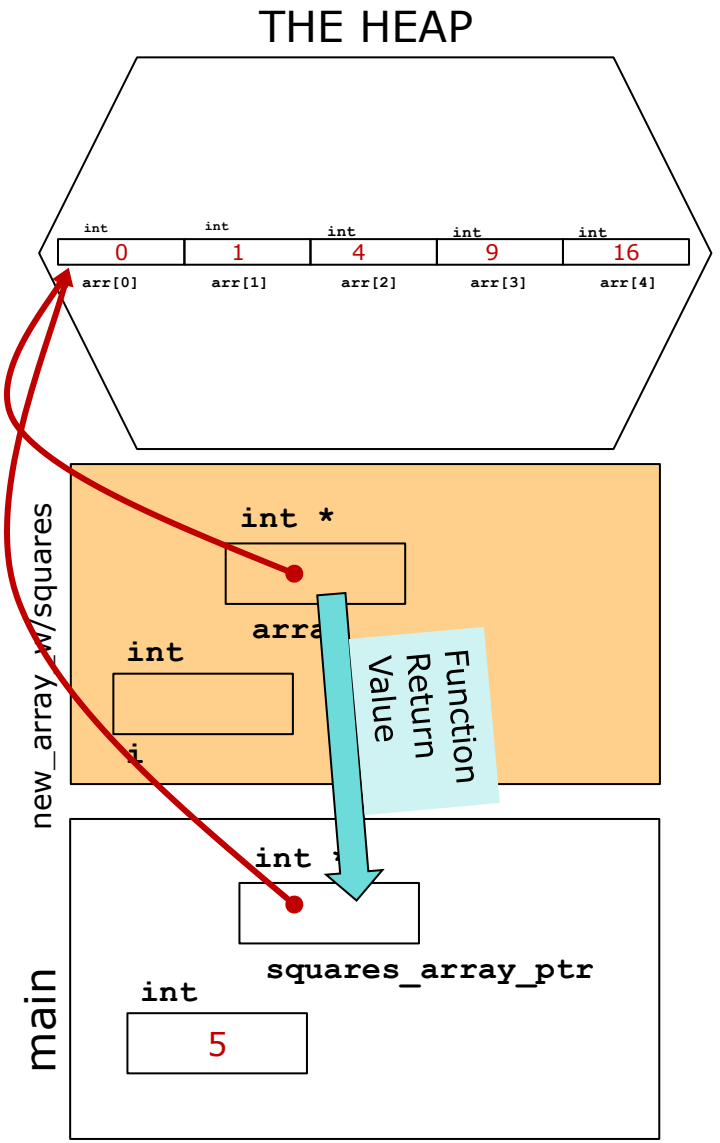
```
int *new_array_with_squares(int size)
{
        int  i;
        int *array = new int [size];

        // 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;
}
```

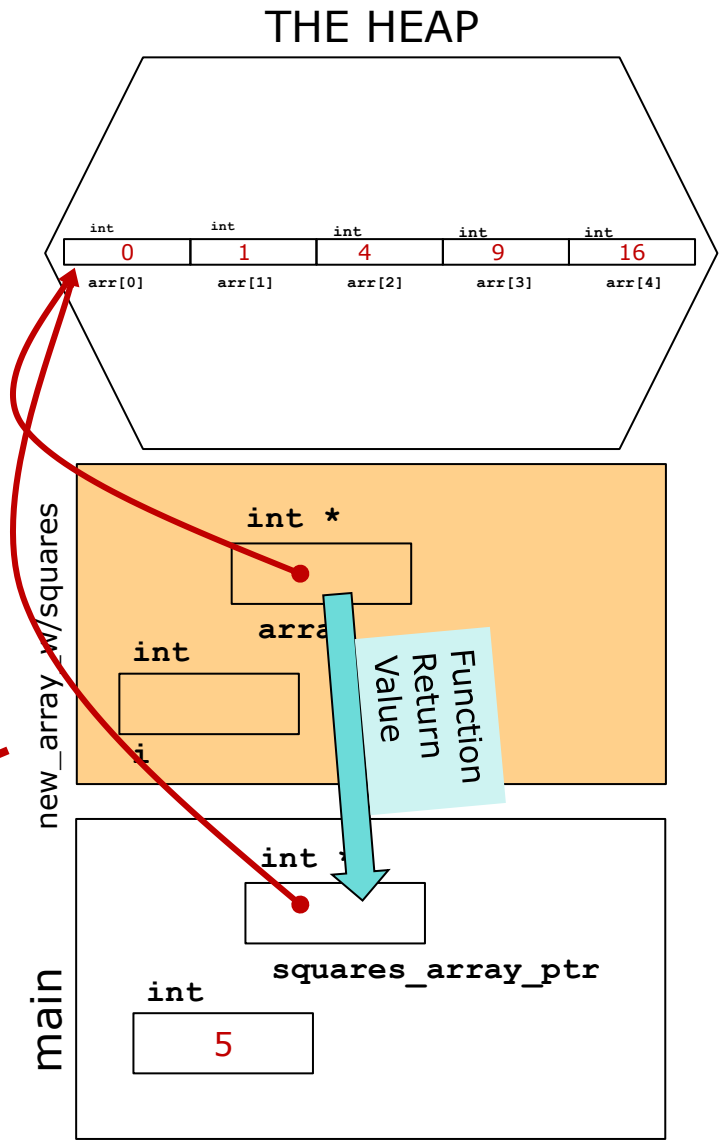
```
int *new_array_with_squares(int size)
{
        int  i;
        int *array = new int [size];

        // 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?

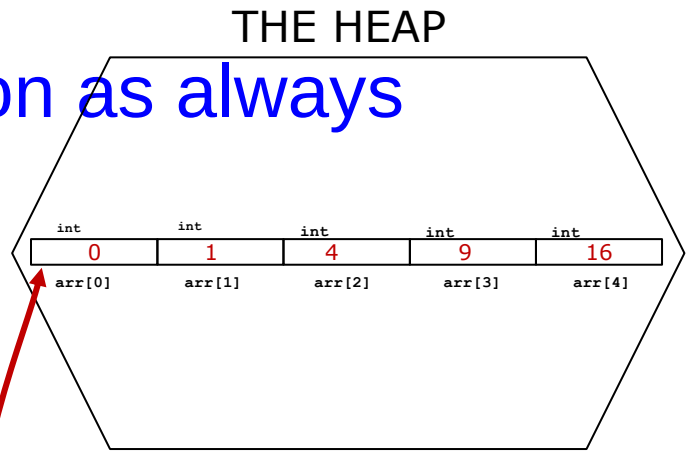
```
int *new_array_with_squares(int size);
void print_array(int *array, int length);
int main ()
{
        int *squares_array_ptr;
        int  len;

        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;
}
```



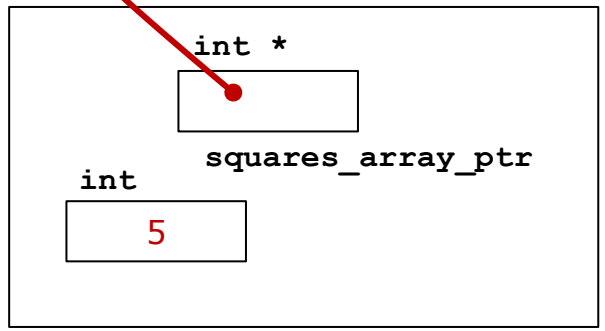
Now we can pass the array to a function as always

```
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;
}
```



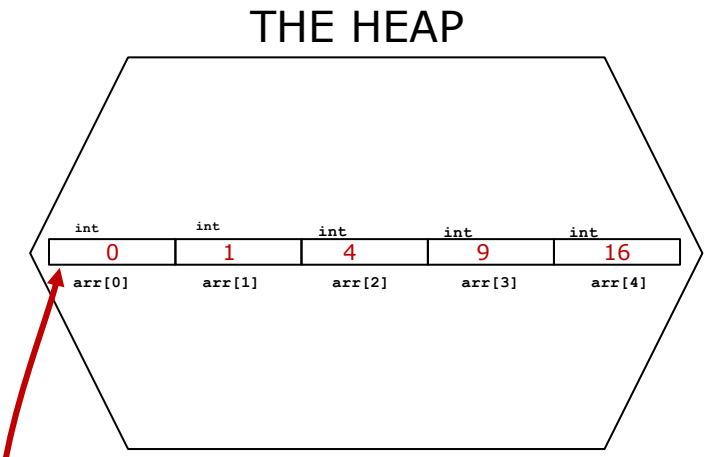
How many squares do you want to store? 5

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



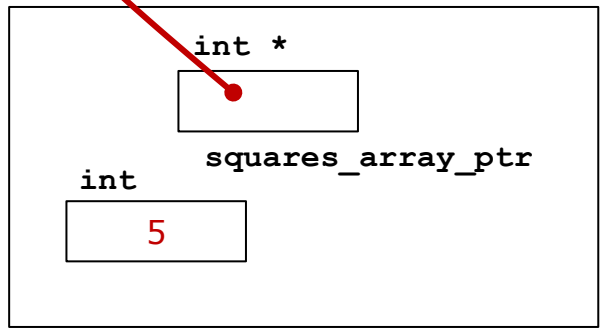
…and use the array until it's deleted!

```
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;
}
```



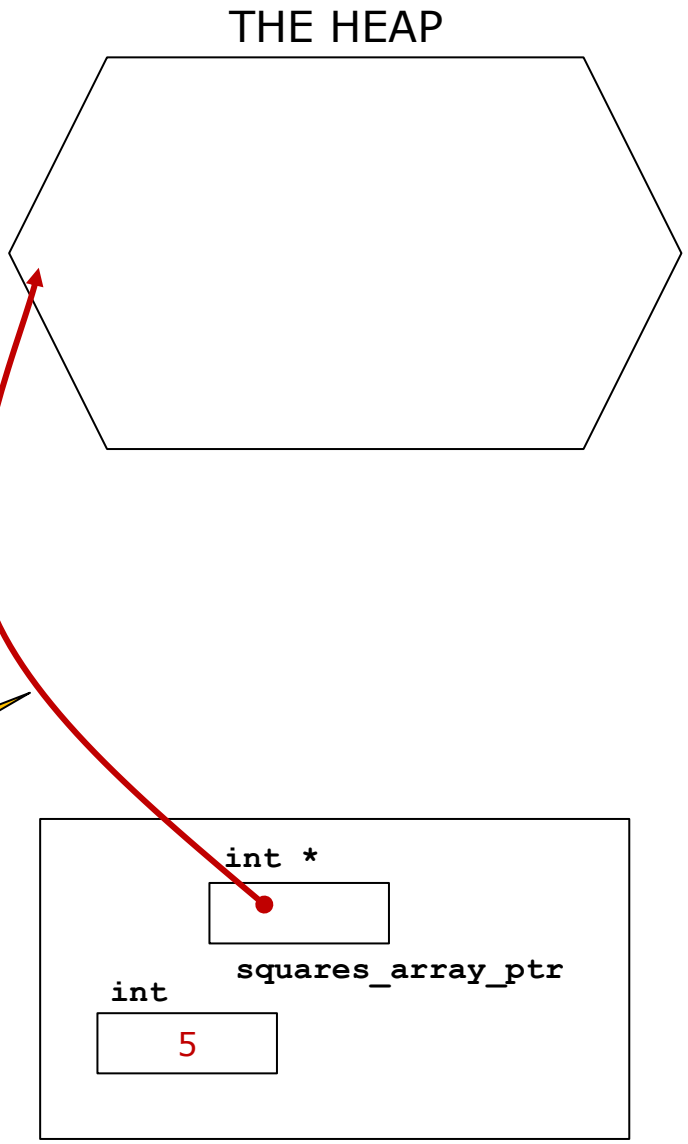
How many squares do you want to store? 5

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



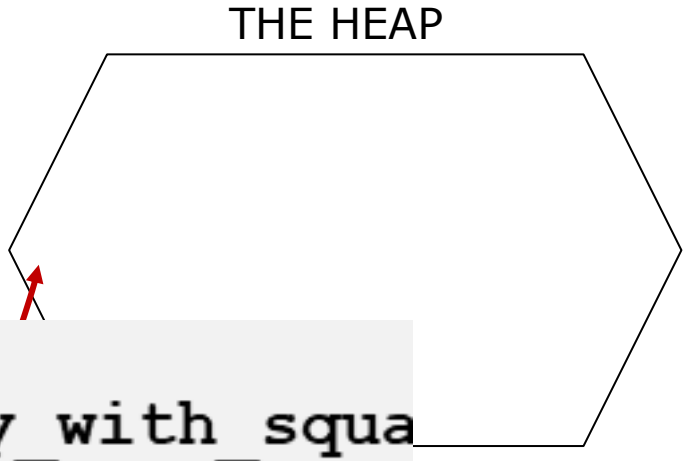
…and use the array until it's deleted!

```
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

…and use the array until it's deleted!




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

IMPORTANT:

[]
Is *required* for deleting arrays
Is *forbidden* when deleting anything else

C++ typically does not give you a nice error message:
your program will behave unpredictably, maybe SEGFAULT, etc.

This is hard to debug if wrong; get it right!

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

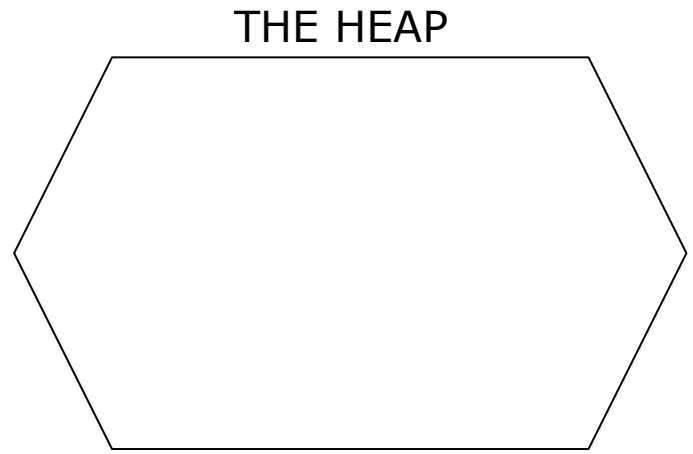
We'll build a dynamic list array!

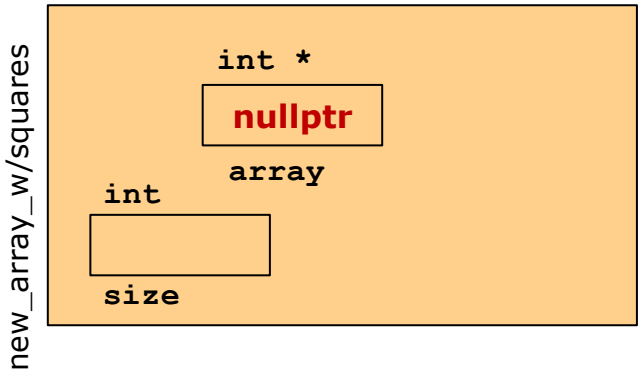
The idea…

- **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!*

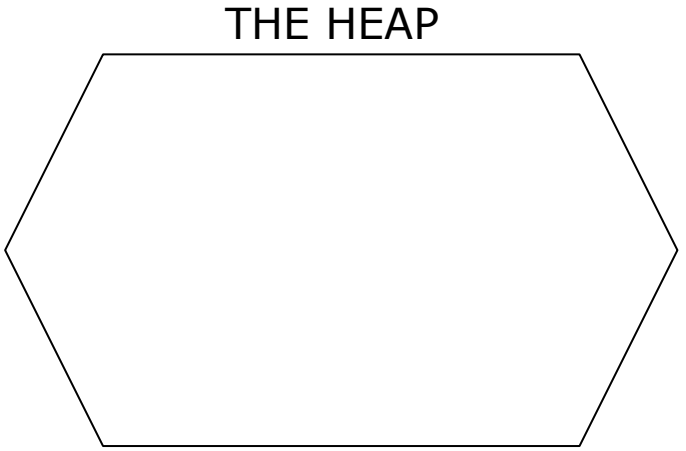
Calling a function that can grow arrays

```
/*
 * 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;
}
```

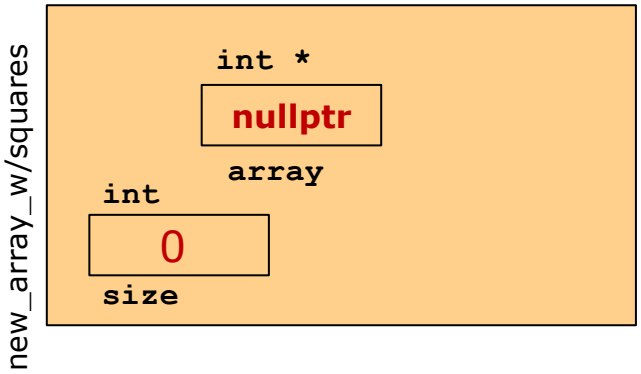




Calling a function that can grow arrays

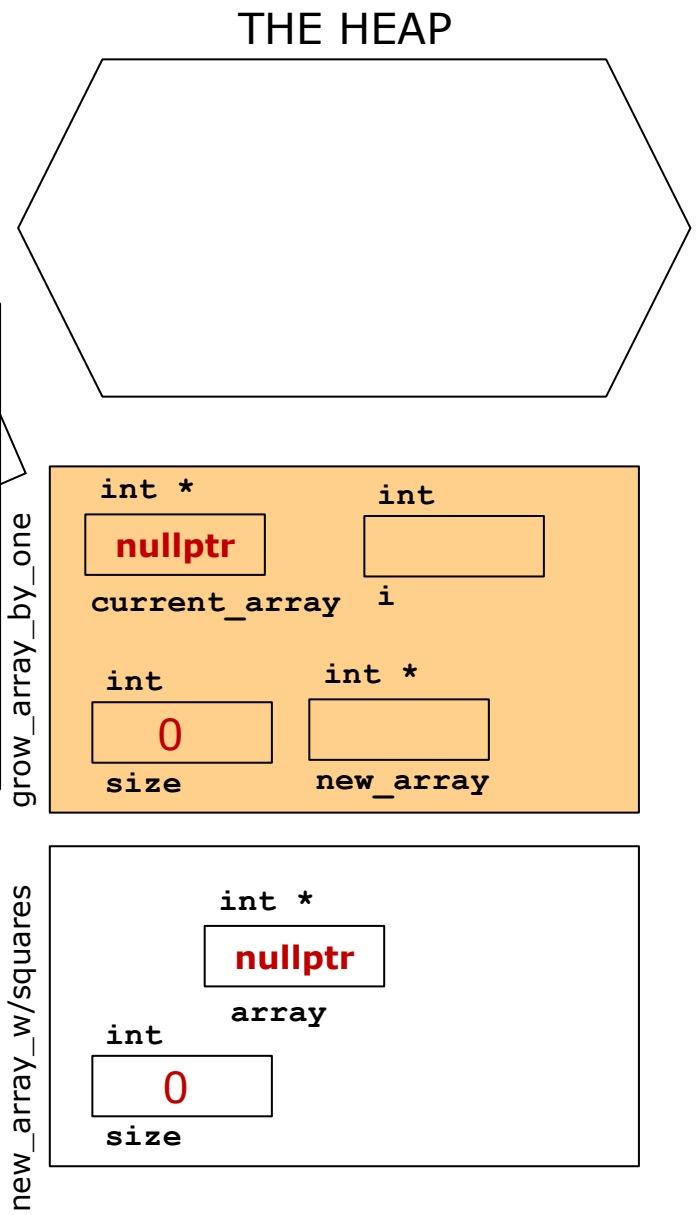



```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // 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;
}
```



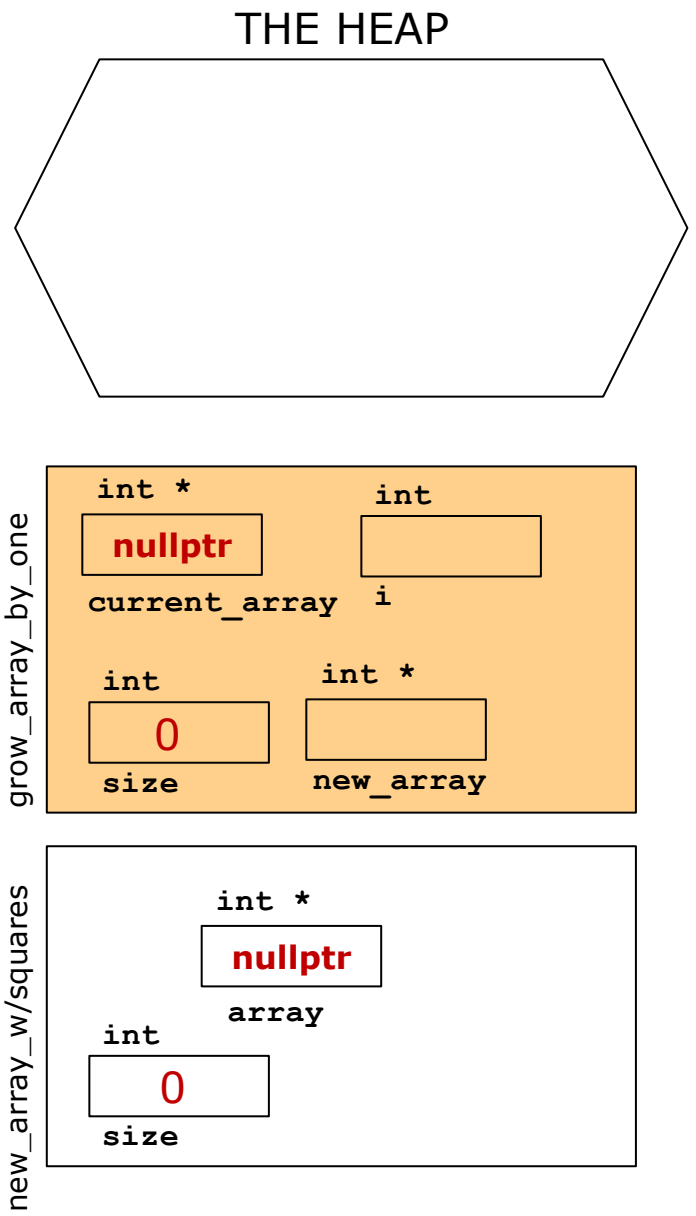
Allocating the first version of the array

```
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;
}
```



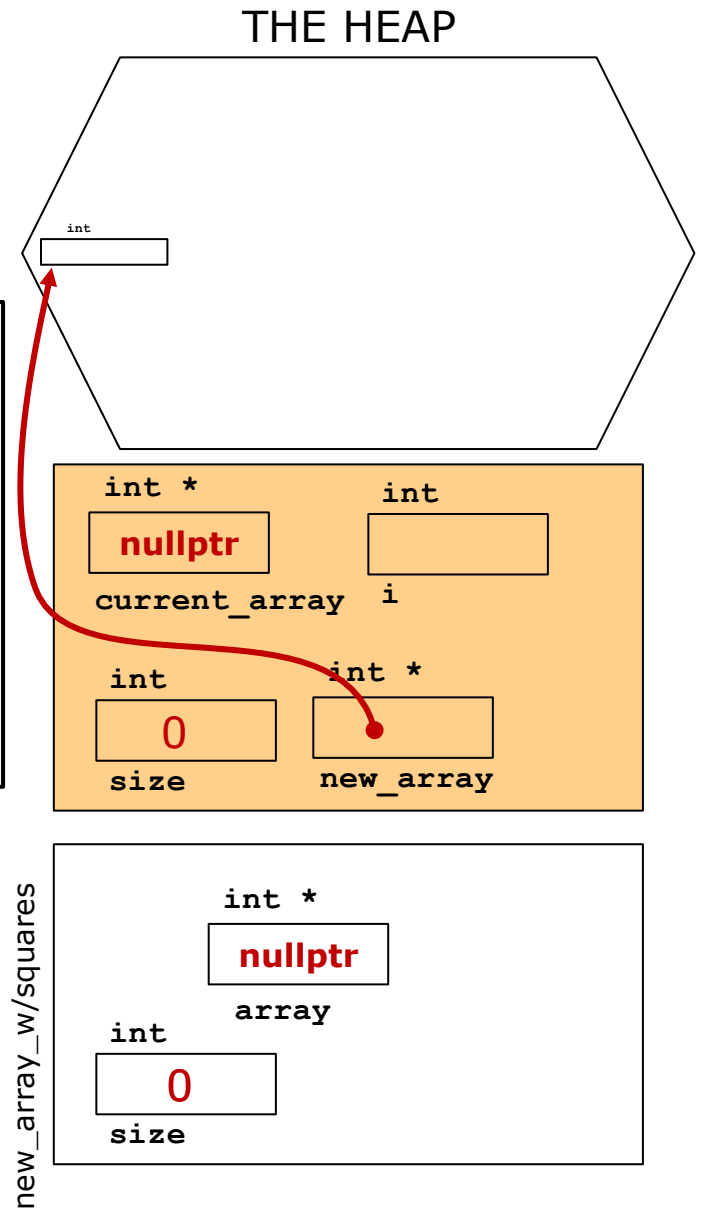
Allocating the first version of the array

```
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;
}
```

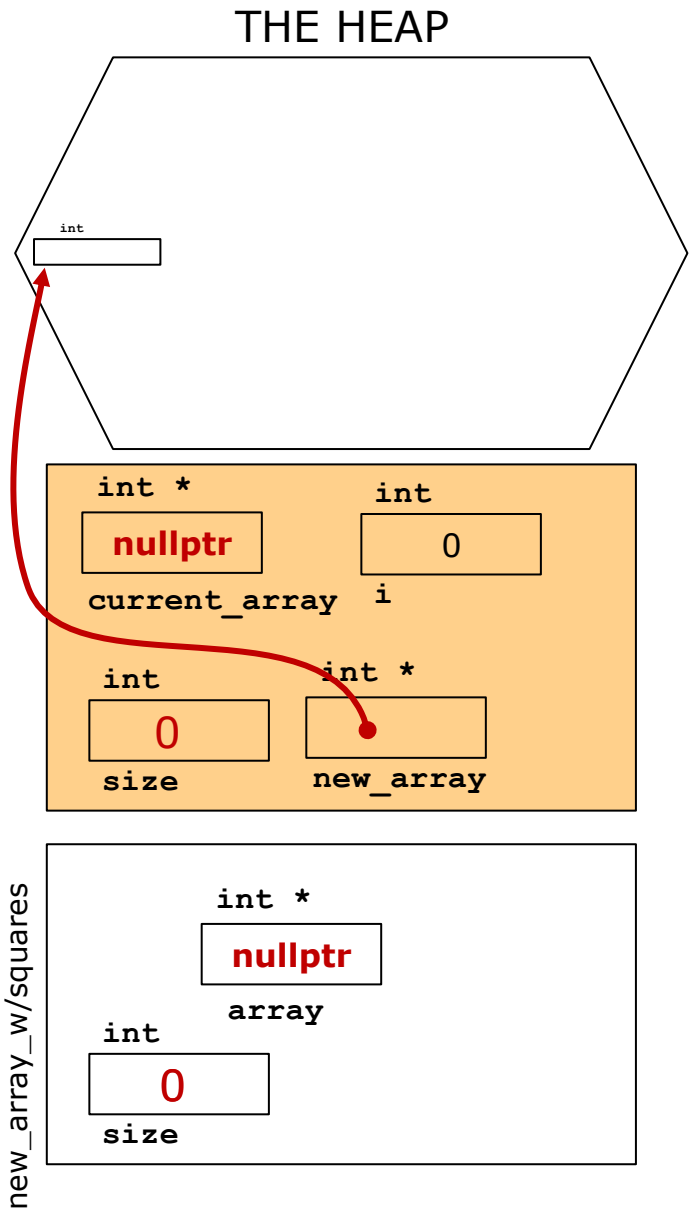


Allocating the first version of the array

```
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)

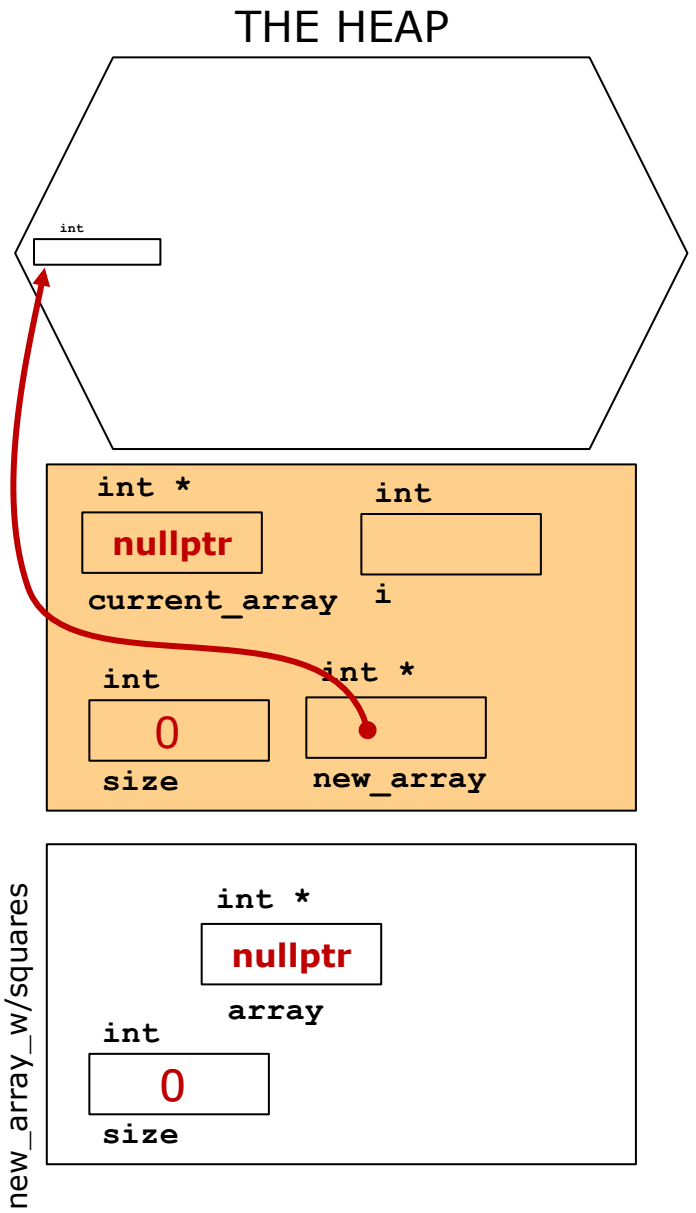


Allocating the first version of the array

```
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;
}
```

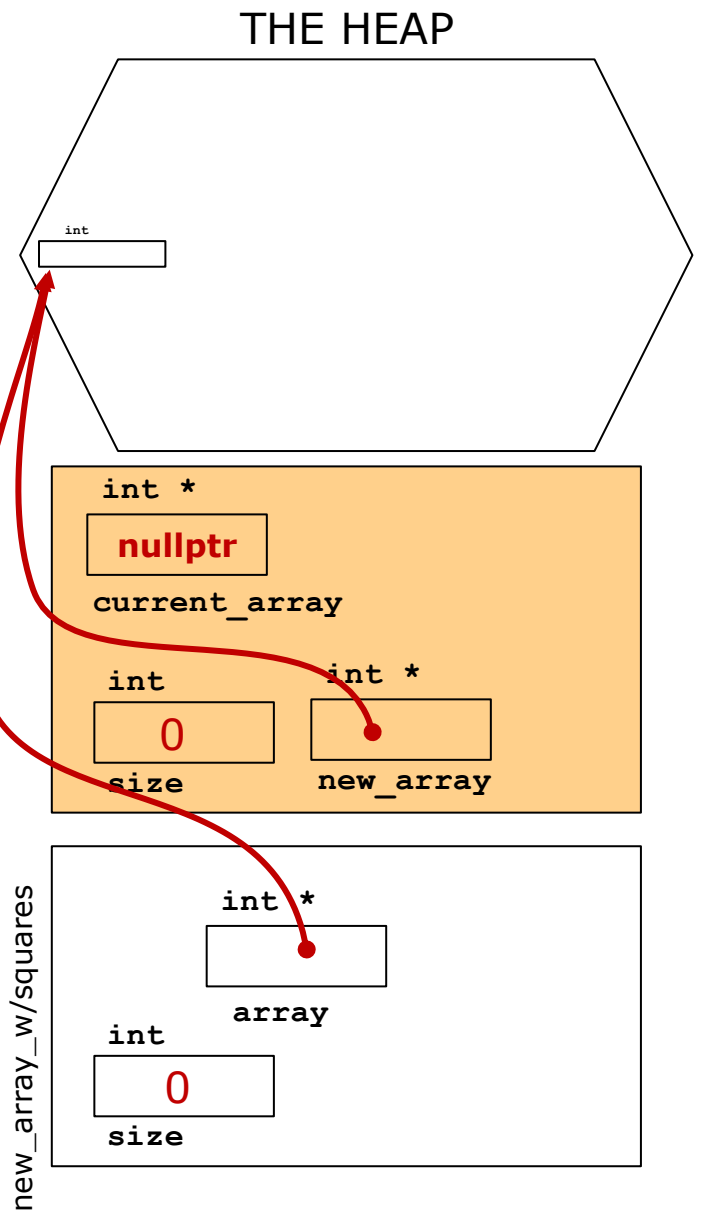
Does nothing this time

(`current_array` is `nullptr`)

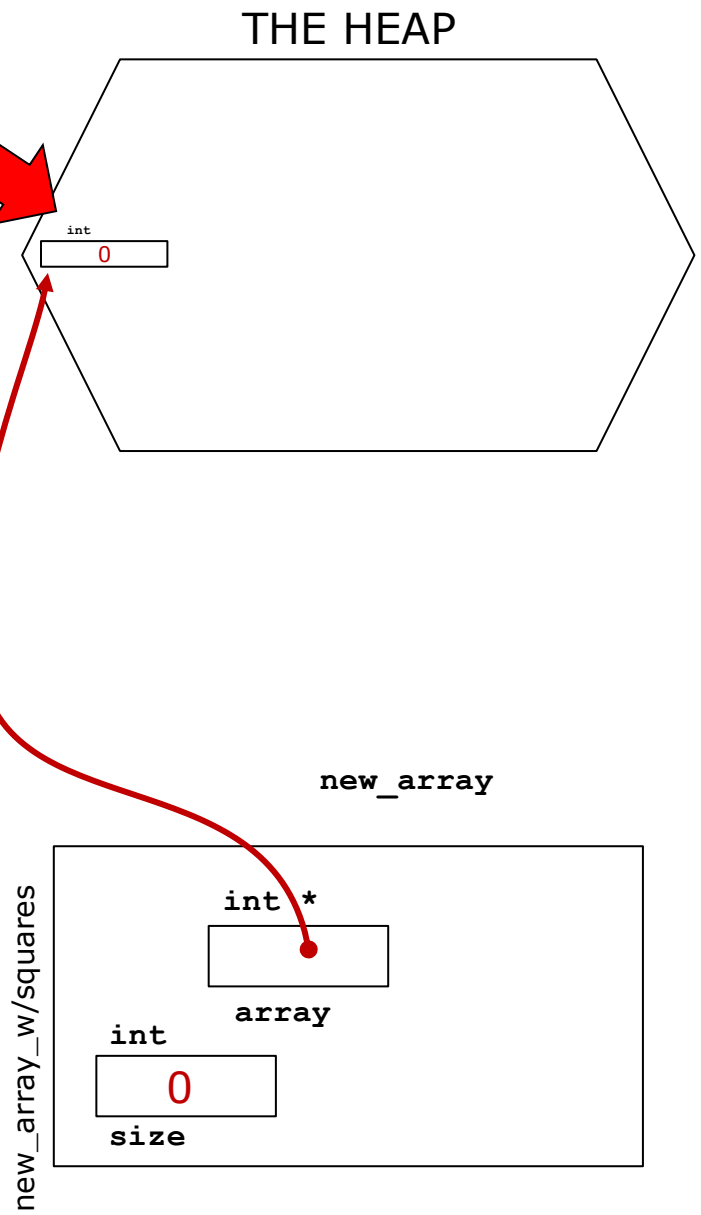


```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // 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;
}
```

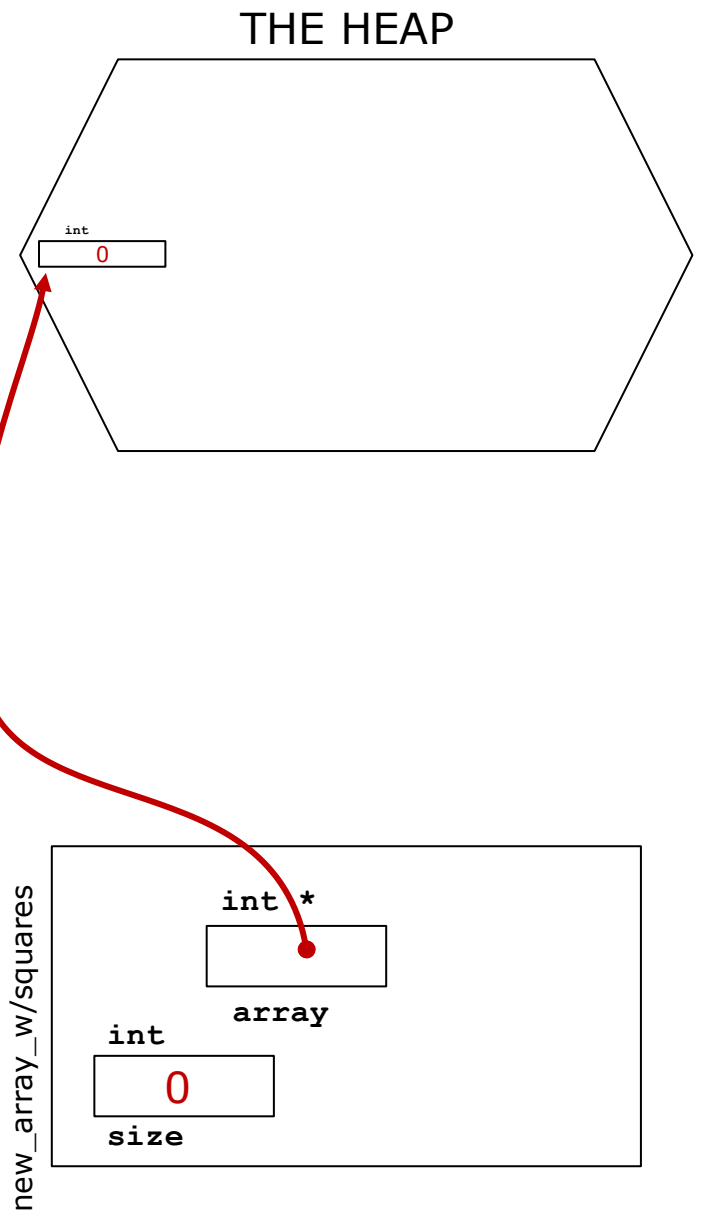


```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // 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;
}
```

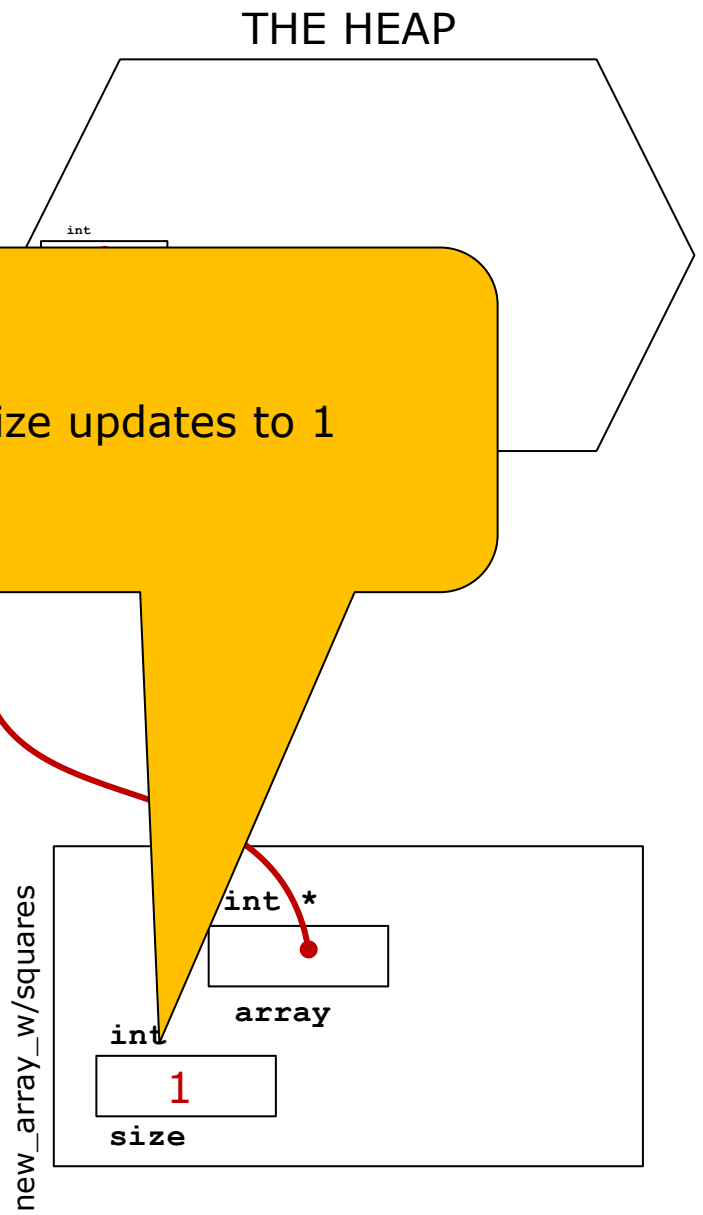
Now let's look at the second time around.



```
/*
 * 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;
}
```

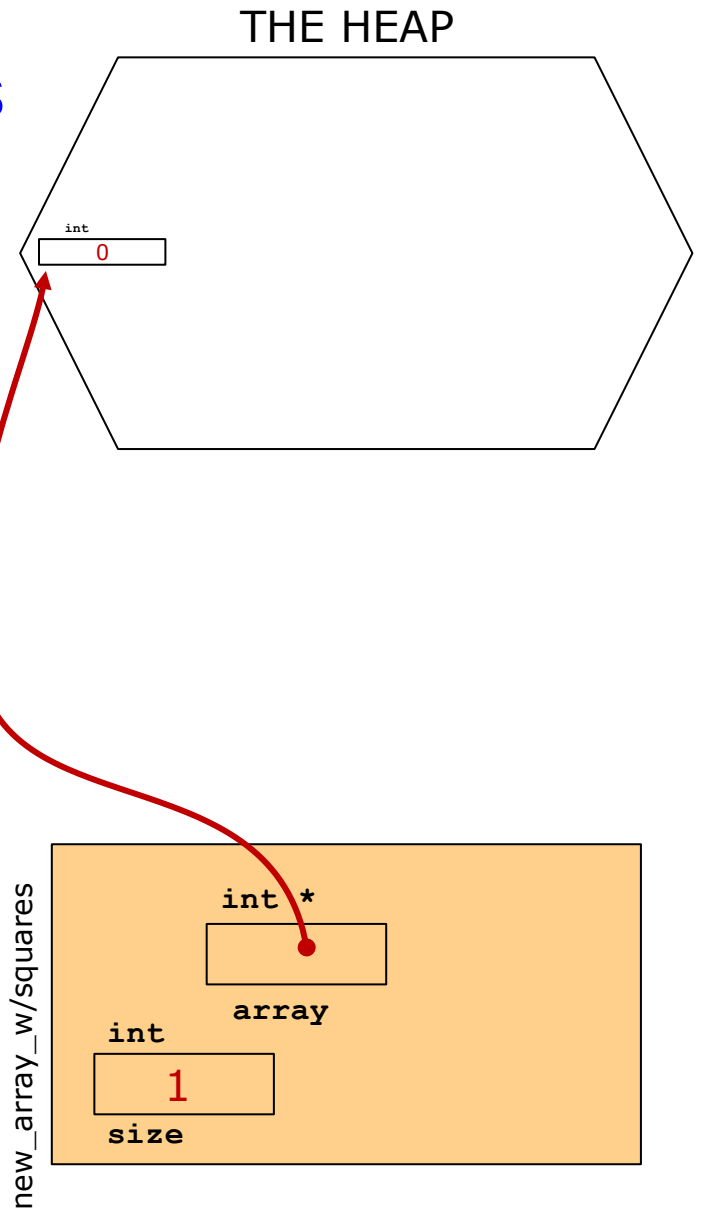
Caller's size updates to 1

Now let's look at the second time around.



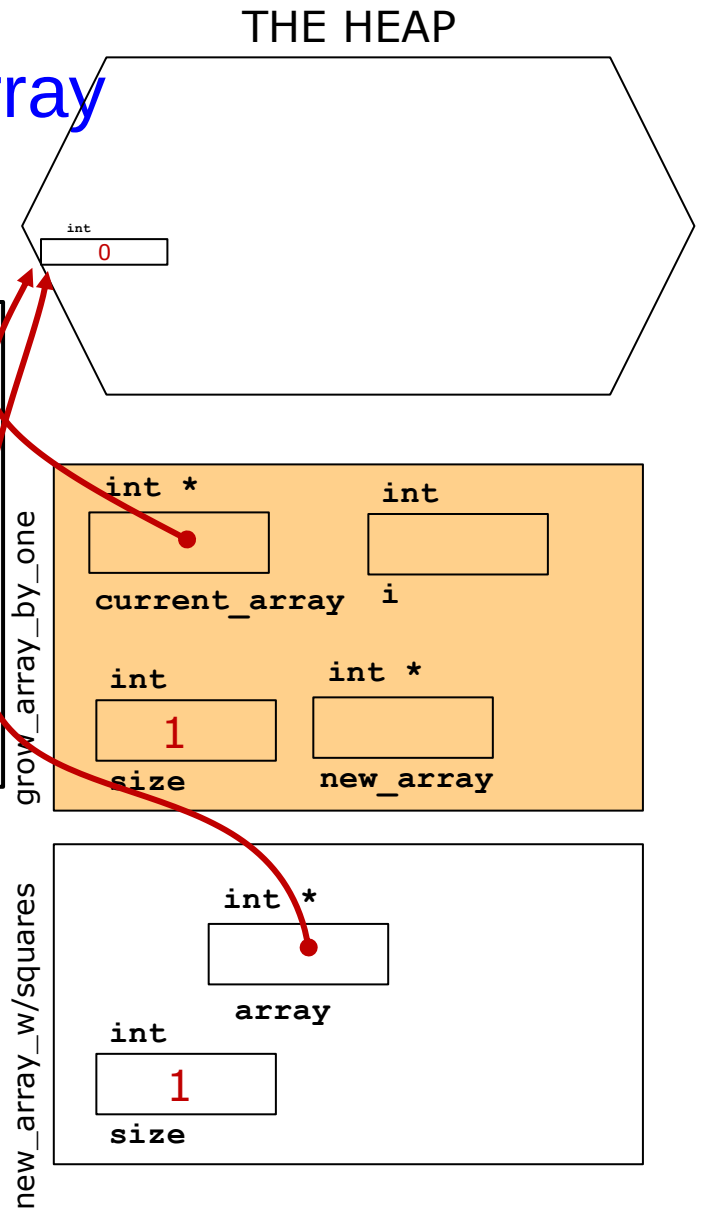
Calling a function that can grow arrays

```
/*
 * 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;
}
```



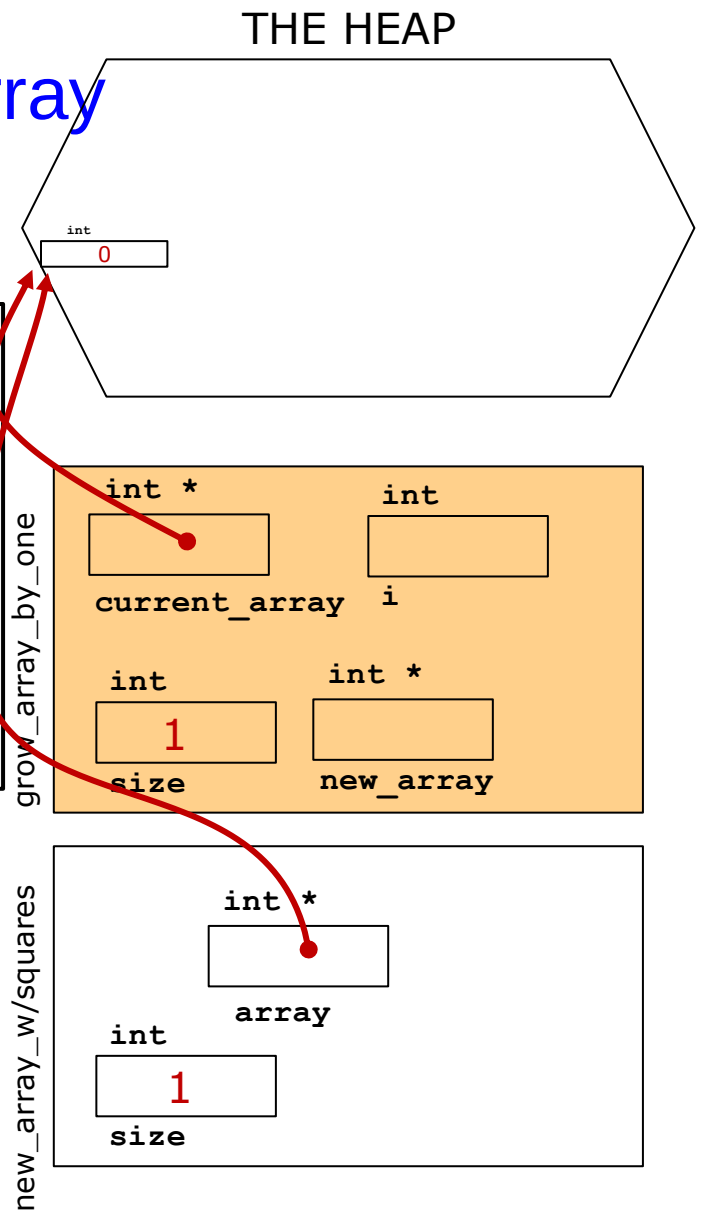
Allocating the *second* version of the array

```
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;
}
```



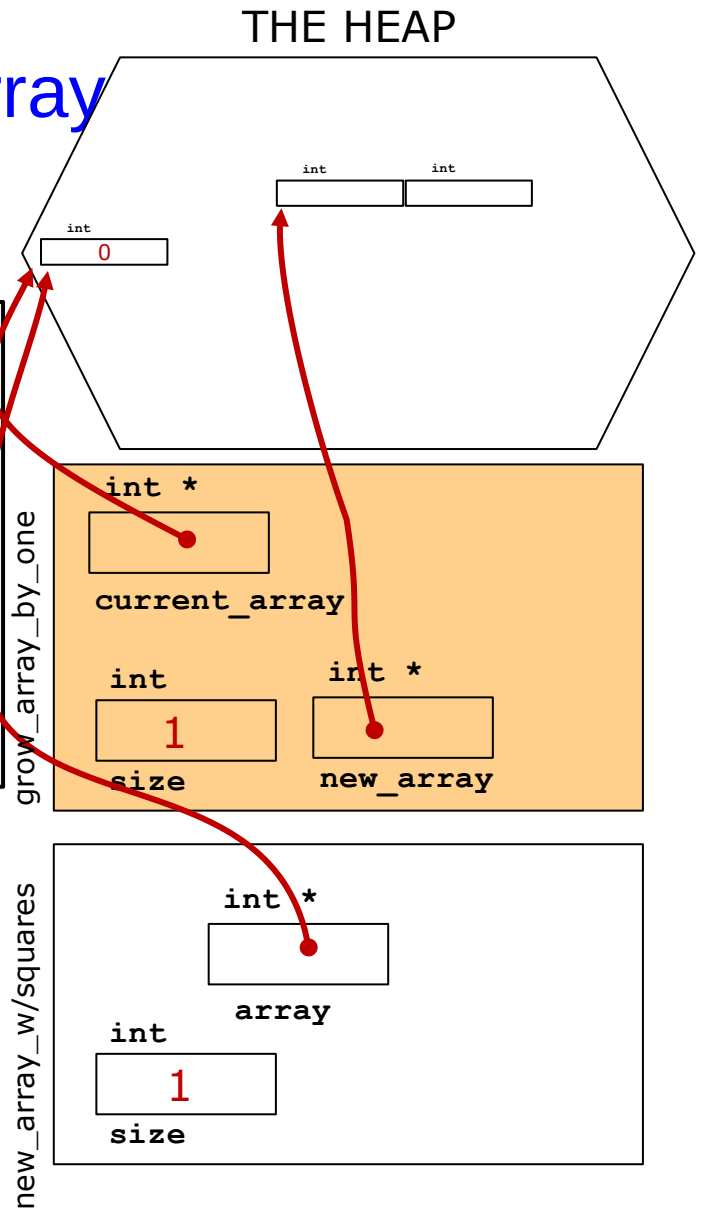
Allocating the *second* version of the array

```
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;
}
```



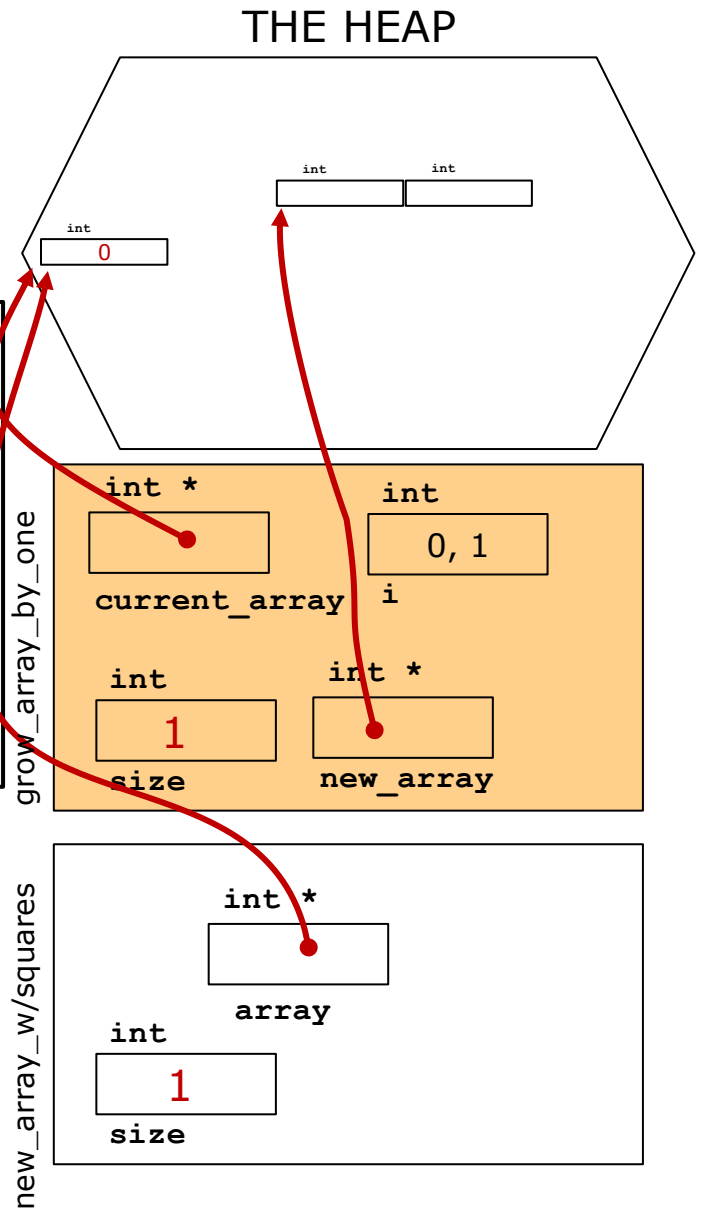
Allocating the *second* version of the array

```
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;
}
```



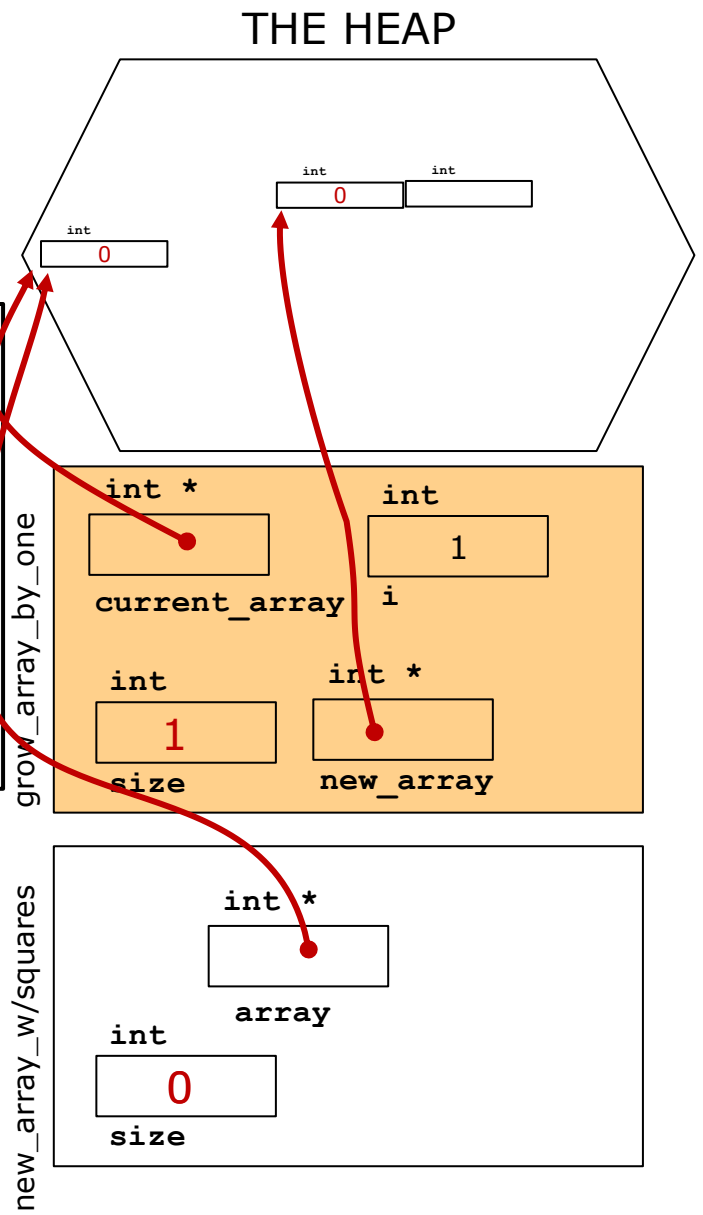
Copying the old to the new

```
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;
}
```



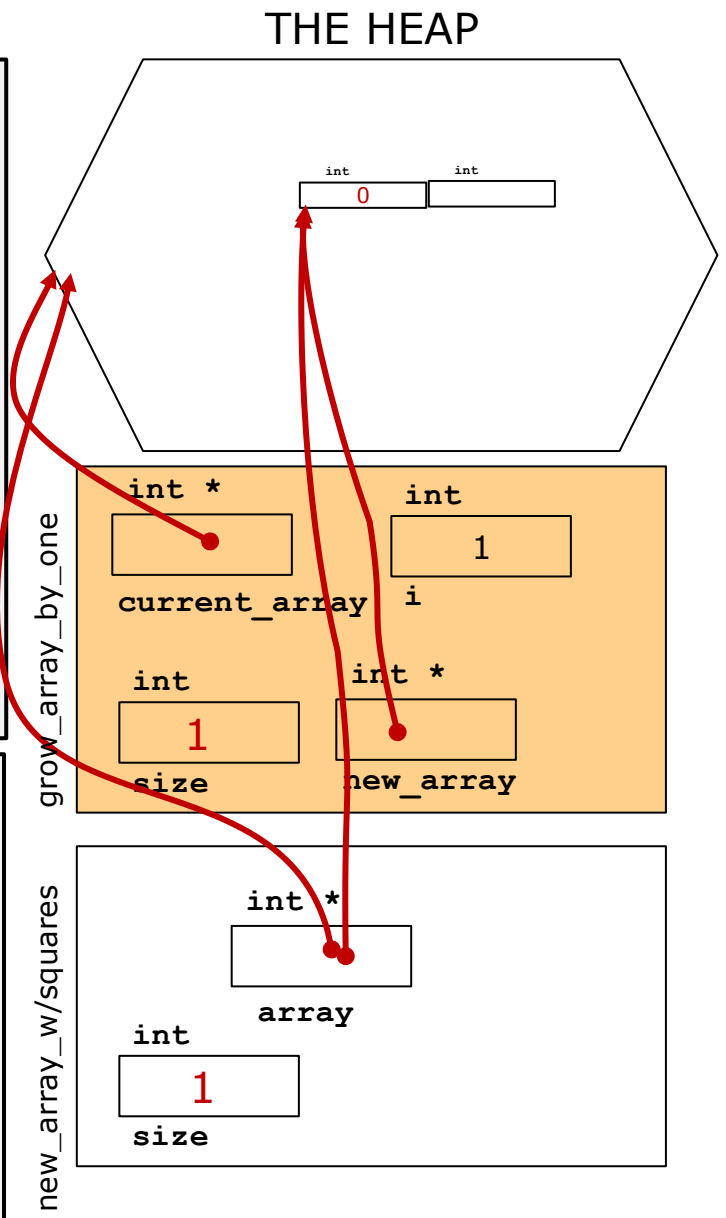
Deleting the one that's been replaced

```
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;
}
```

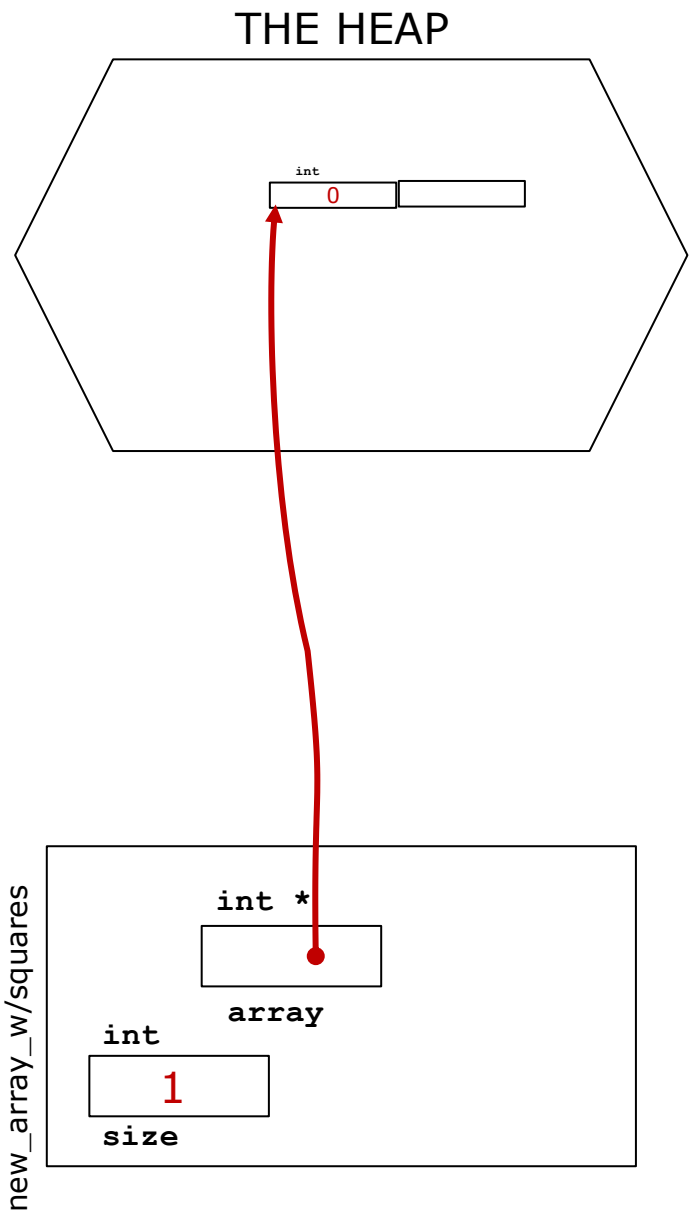


```
/*
 * Get new array on heap and fill each
 * arr[] with i squared
 */
int *new_array_with_squares(int target_size)
{
        int *array = nullptr;

        // 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;
}
```



Moving Towards A Class-based Implementation
Put Array Control Variables in a Struct

A first step: using structs

Data for the array implementation is 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);
}
```

A first step: using structs

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 functions

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

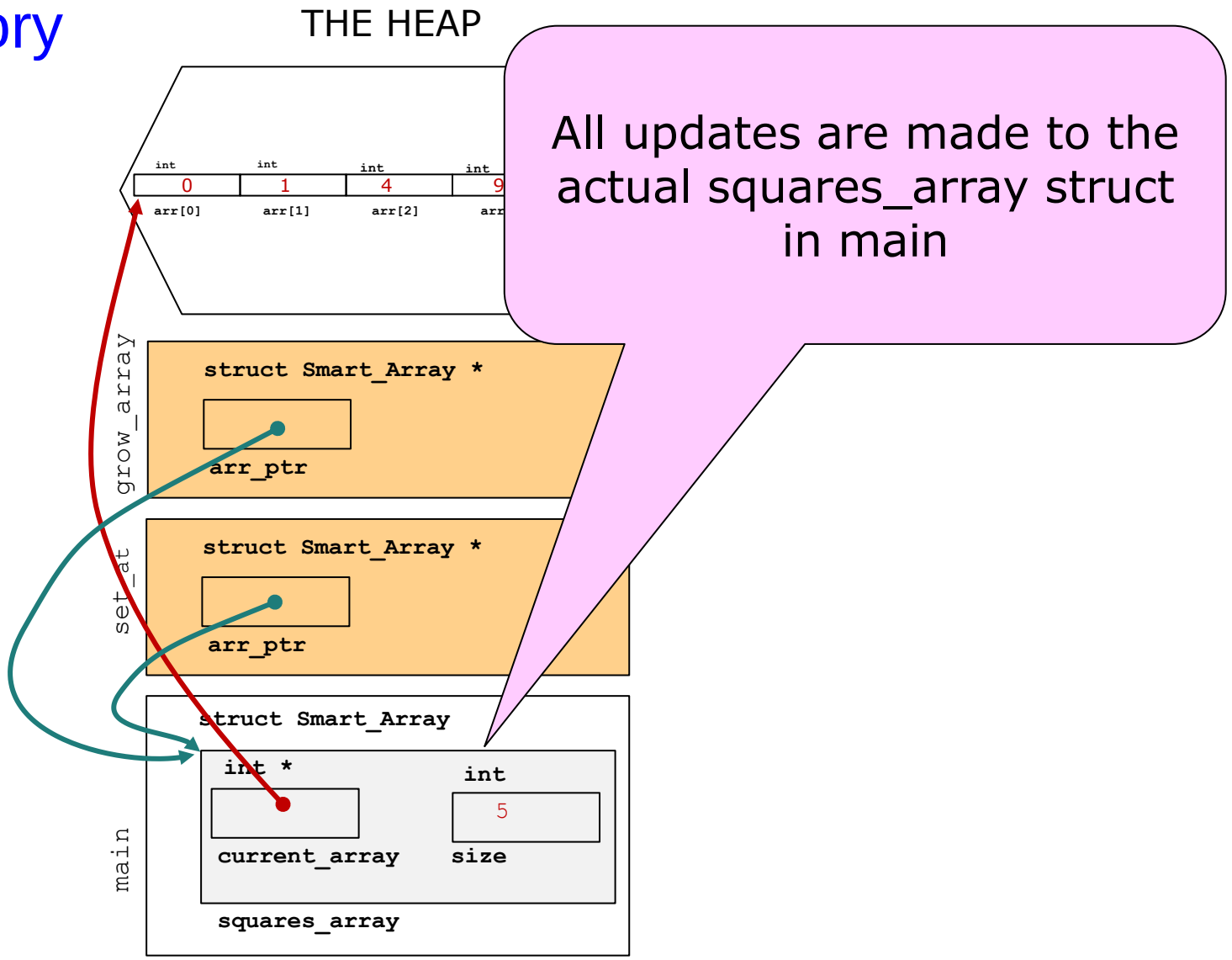
Example of two of the Smart array functions

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

```
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;
}
```

..and for the same reason we pass in the pointer to grow_array, which is actually manipulating the struct that lives in main's activation record.

What's in memory



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;
}
```

The next step: Wrapping it as a class

The Smart_Array Class

Public or private?

```
Class Smart_Array {


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

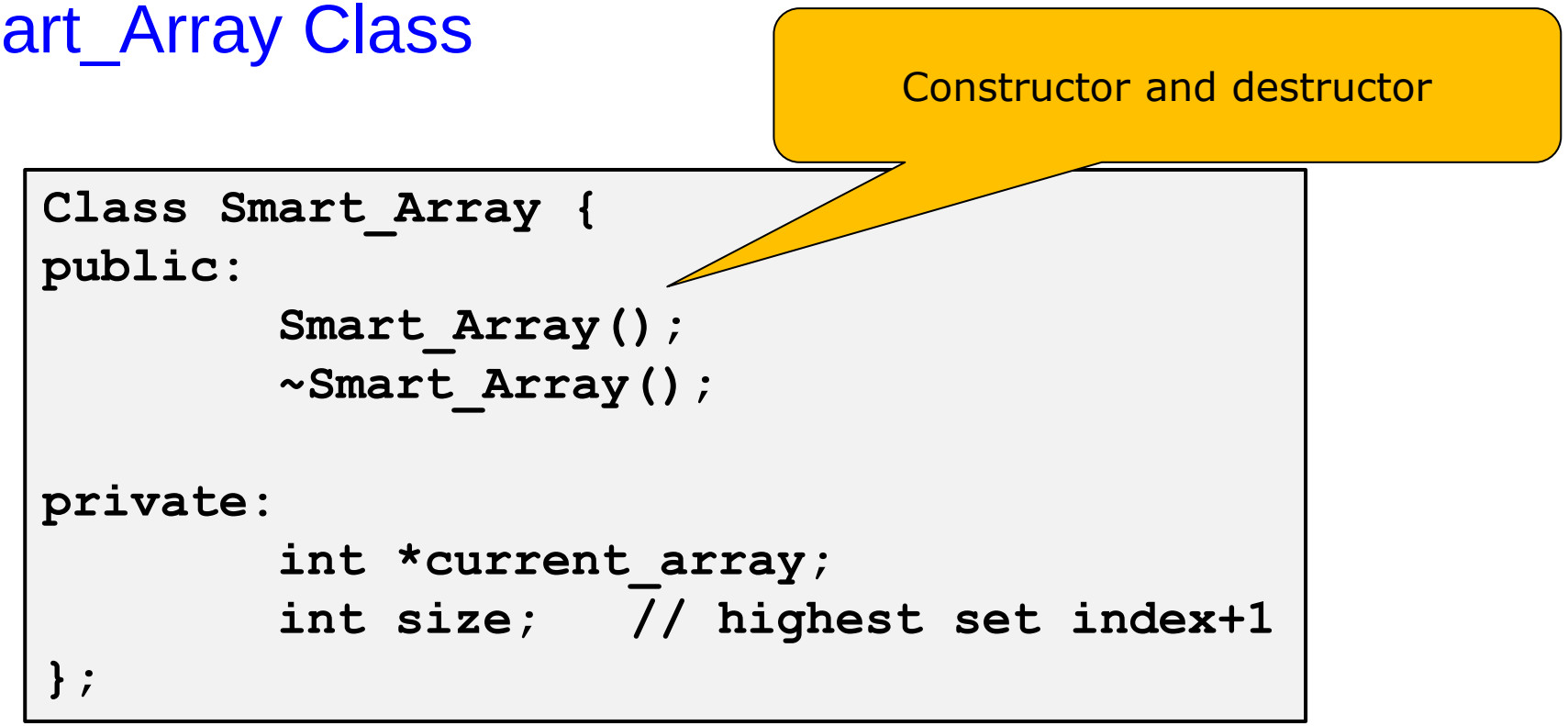
The Smart_Array Class

What else are we missing?

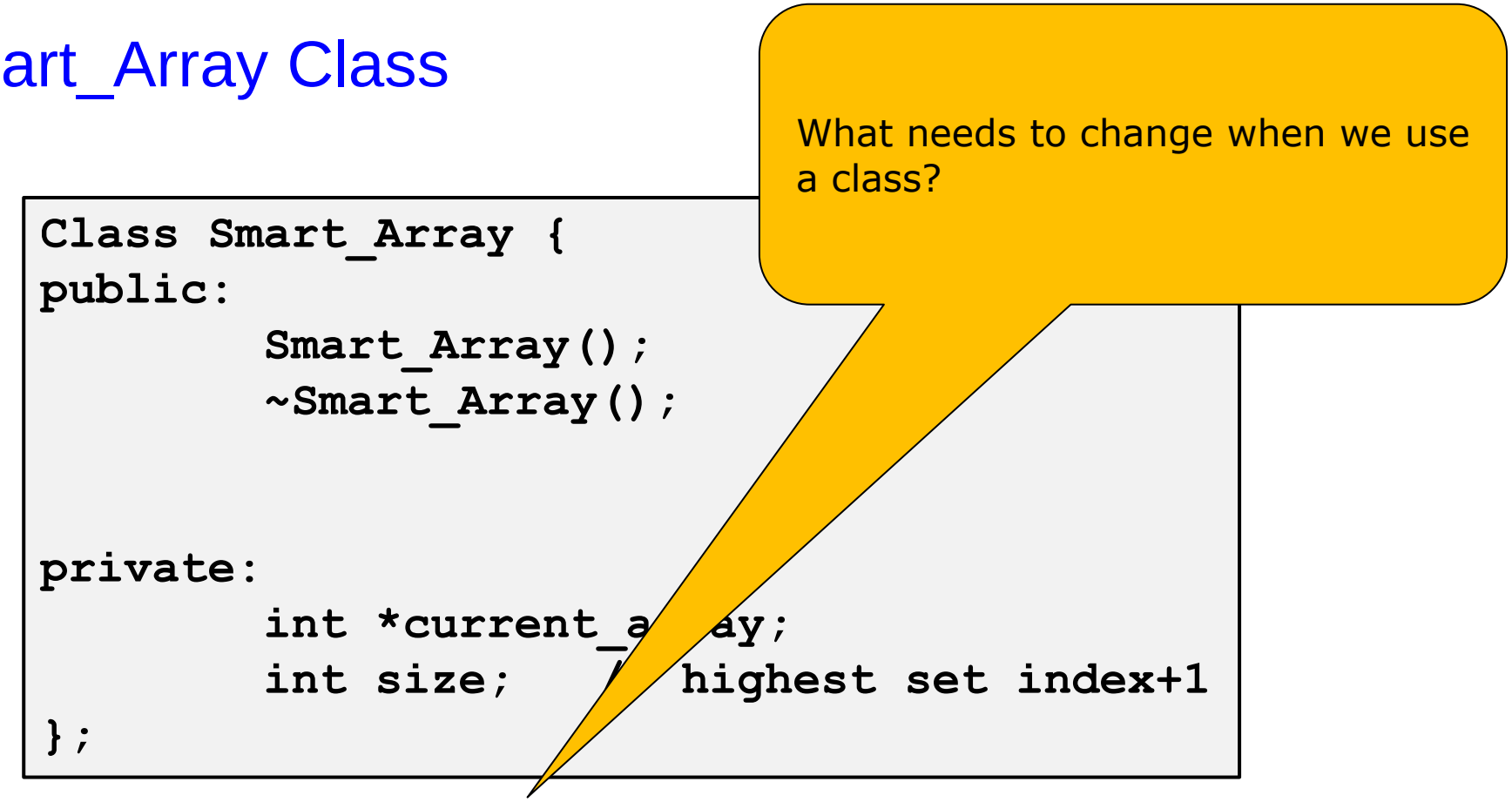
With classes, users don't see the data.

```
Class Smart_Array {

private:
        int *current_array;
        int size;   // highest set index+1
};
```

The Smart_Array Class



The Smart_Array Class



The old set_at/get_at functions:

```
void set_at(Smart_Array *arr_ptr, int i, int new_value)
int get_at(Smart_Array arr, int i);
```

The Smart_Array Class

```
Class Smart_Array {
public:
        Smart_Array();
        ~Smart_Array();
        void set_at(int i, int new_value);
        int get_at(int i);

private:
        int *current_array;
        int size;   // highest set index+1
};
```

The old set_at/get_at functions:

```
void set_at(Smart_Array *arr_ptr, int i, int new_value)
int get_at(Smart_Array arr, int i)
```

The Smart_Array Class

With classes, we don't have to pass the pointer explicitly...C++ knows which instance is to be referenced.

```
Class Smart_Array {
public:
        Smart_Array();
        ~Smart_Array();
        void set_at(int i, int new_value);
        int get_at(int i);

private:
        int *current_array;
        int size;   // highest set index+1
};
```

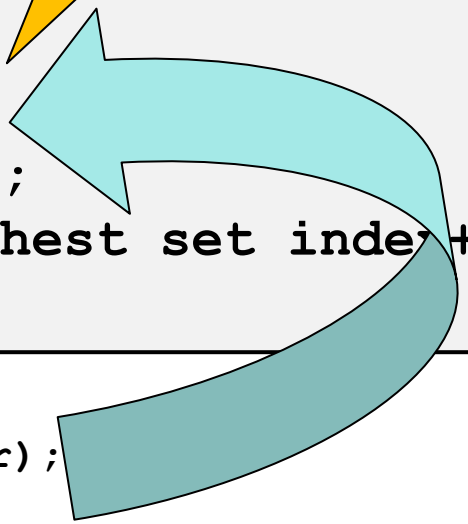
The old set_at/get_at functions:

```
void set_at(Smart_Array *arr_ptr, int i, int new_value)
int get_at(Smart_Array arr, int i)
```

The Smart_Array Class

```
Class Smart_Array {
public:
        Smart_Array();
        ~Smart_Array();
        void set_at(int i, i    new_value);
        int get_at(int i);
private:
        void grow_array();
        int *current_array;
        int size;   // highest set inde +1
};
```

Private:
Users see an array that grows automatically as needed.



The old grow_array function:

```
void grow_array(Smart_Array *arr_ptr);
```

From Arrays to Lists

What the user sees

```
Class Smart_Array {
public:
        Smart_Array();
        ~Smart_Array();
        void set_at(int i, int new_value;
        int get_at(int i);
};
```

```
        void grow_array();
        int *current_array;
        int size;   // highest set index+1
};
```

What the user sees

```
Class Smart_Array {
public:
        Smart_Array();
        ~Smart_Array();
        void set_at(int i, int new_value;
        int get_at(int i);
};
```

HEY!

That's a pretty good start on an interface to *any* list, not just to an array!

List interface: what other functions might we want?

```
Class List {
public:
        List();
        ~List();

        // By index
        void set_at(int i, int new_value;
        int get_at(int i);
};
```

A Fancier List Interface

```
Class List {
public:
        List();
        ~List();

        // By index
        void set_at(int i, int new_value;
        int get_at(int i);

        // At front
        int removeFirst();
        void addToFront(int n);




};
```

A Fancier List Interface

```
Class List {
public:
        List();
        ~List();

        // By index
        void set_at(in
        int get_at(int i);

        // At front
        int removeFirst();
        void addToFront(int n);




};
```

Remove the first item…

…leaving us with a *new* first item!

A Fancier List Interface

```
Class List {
public:
        List();
        ~List();

        // By index
        void set_at(int
        int get_at(int i);

        // At front
        int removeFirst();
        void addToFront(int n);




};
```

Put a new item at the front..

…bumping back all the others!!

A Fancier List Interface

```
Class List {
public:
        List();
        ~List();

        // By index
        void set_at(i
        int get_at(ir

        // At front
        int removeFirst(
        void addToFront    n);

        // At back
        void addToBack(int n);
};
```

Append a new item to the *end* of the list.

A Fancier List Interface

```
Class List {
public:
        List();
        ~List();

        // By index
        void set_at(int i, int new_value;
        int get_at(int i);

        // At front
        int removeFirst();
        void addToFront(int n);

        // At back
        void addToBack(int n);
        void addAtPostition(int n, int position);
};
```

A Fancier List Interface

```
Class List {
public:
        List();
        ~List();
```

Many other variations on the List interface are possible.

```
        // At back
        void addToBack(int n);
};
```

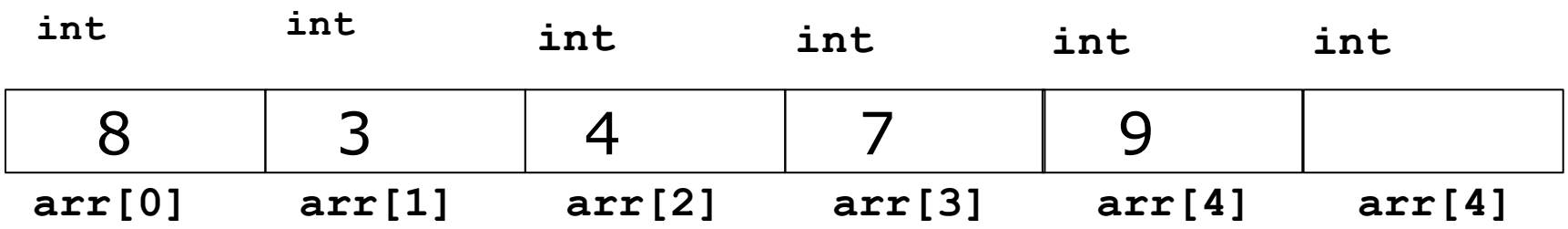
Review: what we've just seen

- **Lists are a very useful data abstraction**
- **Typical list interfaces let you:**
  - Add items to the list at the beginning and/or end
  - Remove items from the beginning and/or end
  - Get to all the items by index in the current list
  - Sometimes:
    - Remove items from the middle of the list
    - Insert items in the middle of the list
    - Etc.
- **NOW WE WILL EXPLORE AN INTERESTING IDEA: ARE THERE DIFFERENT WAYS OF PROVIDING THIS SAME LIST MANAGEMENT SERVICE?**

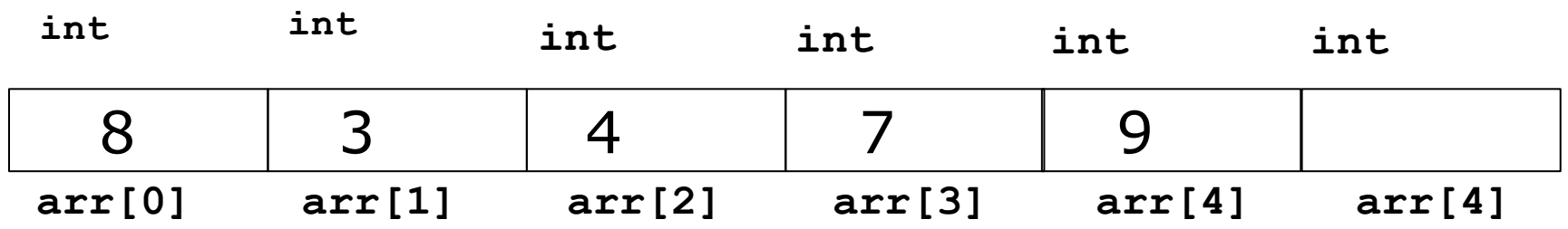
DISCUSSION

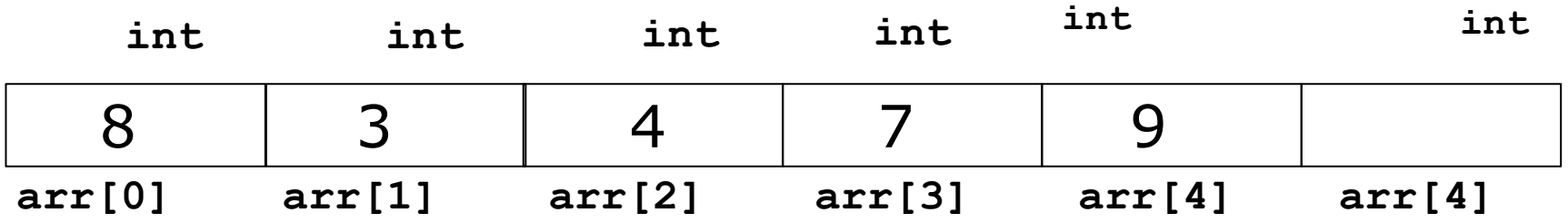
What would happen if we tried to add these new capabilities using our array-based implementation?

Removing from the front of an array-based list

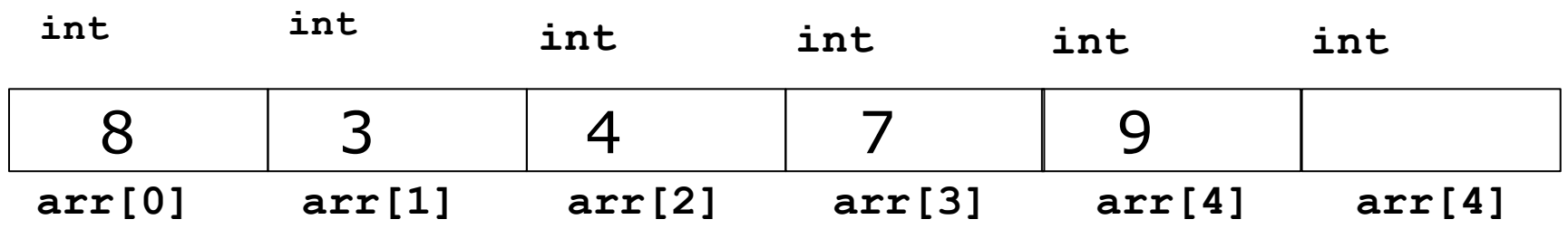


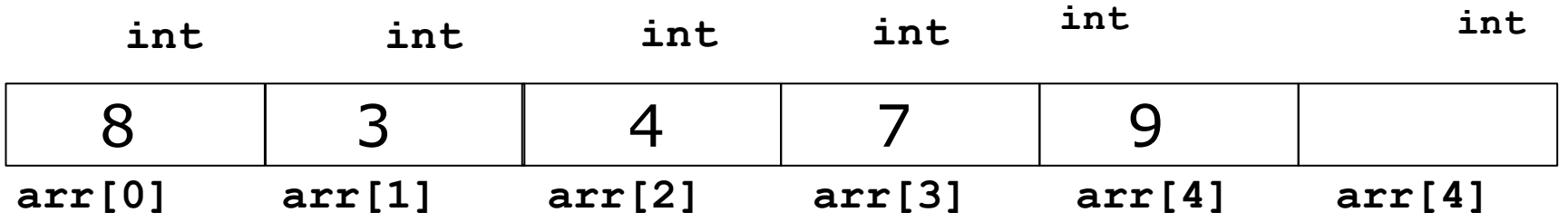
Removing from the front of an array-based list





Adding at the end of an array-based list





99

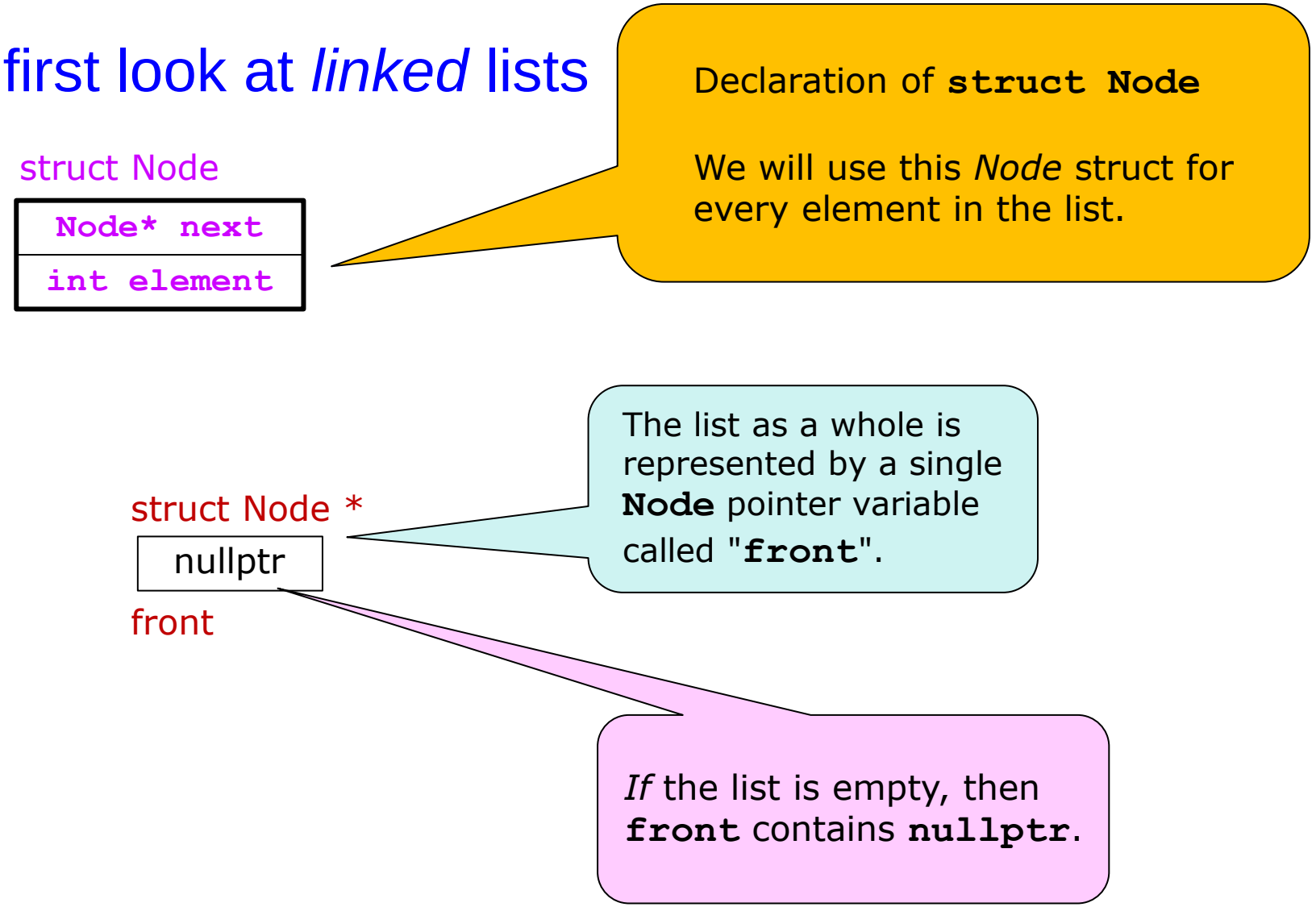
Implementing lists with arrays

- **Adding at the end is:**
  - Easy?
  - Hard?
- **Adding at the beginning is:**
  - Easy?
  - Hard?
- **Getting a value by index is:**
  - Easy?
  - Hard?
- **Deleting at the end is:**
  - Easy?
  - Hard?
- **Deleting at the beginning is:**
  - Easy?
  - Hard?
- **Inserting / deleting in the middle is:**
  - Easy?
  - Hard?

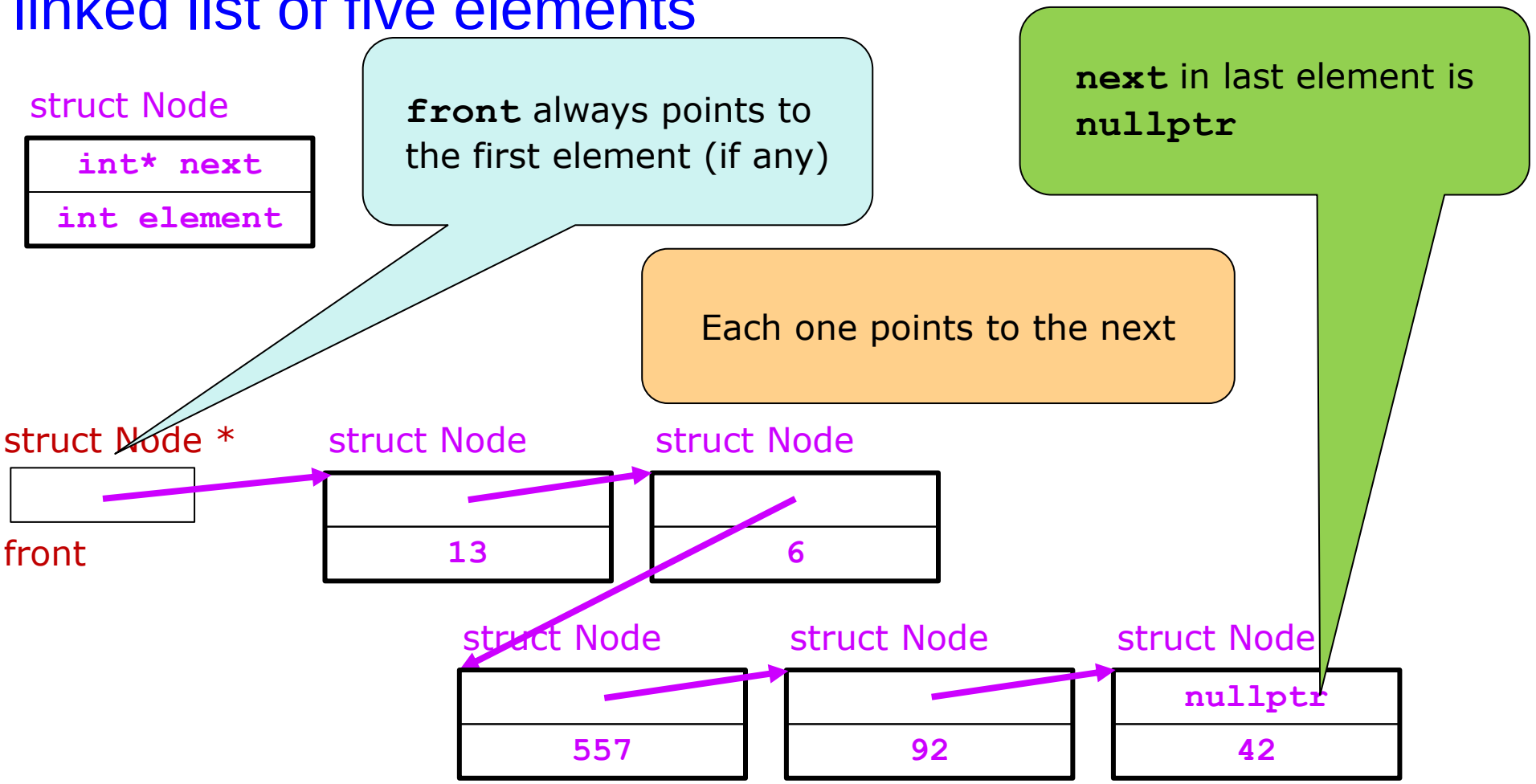
DISCUSSION

Are there other ways we could implement the same interface?

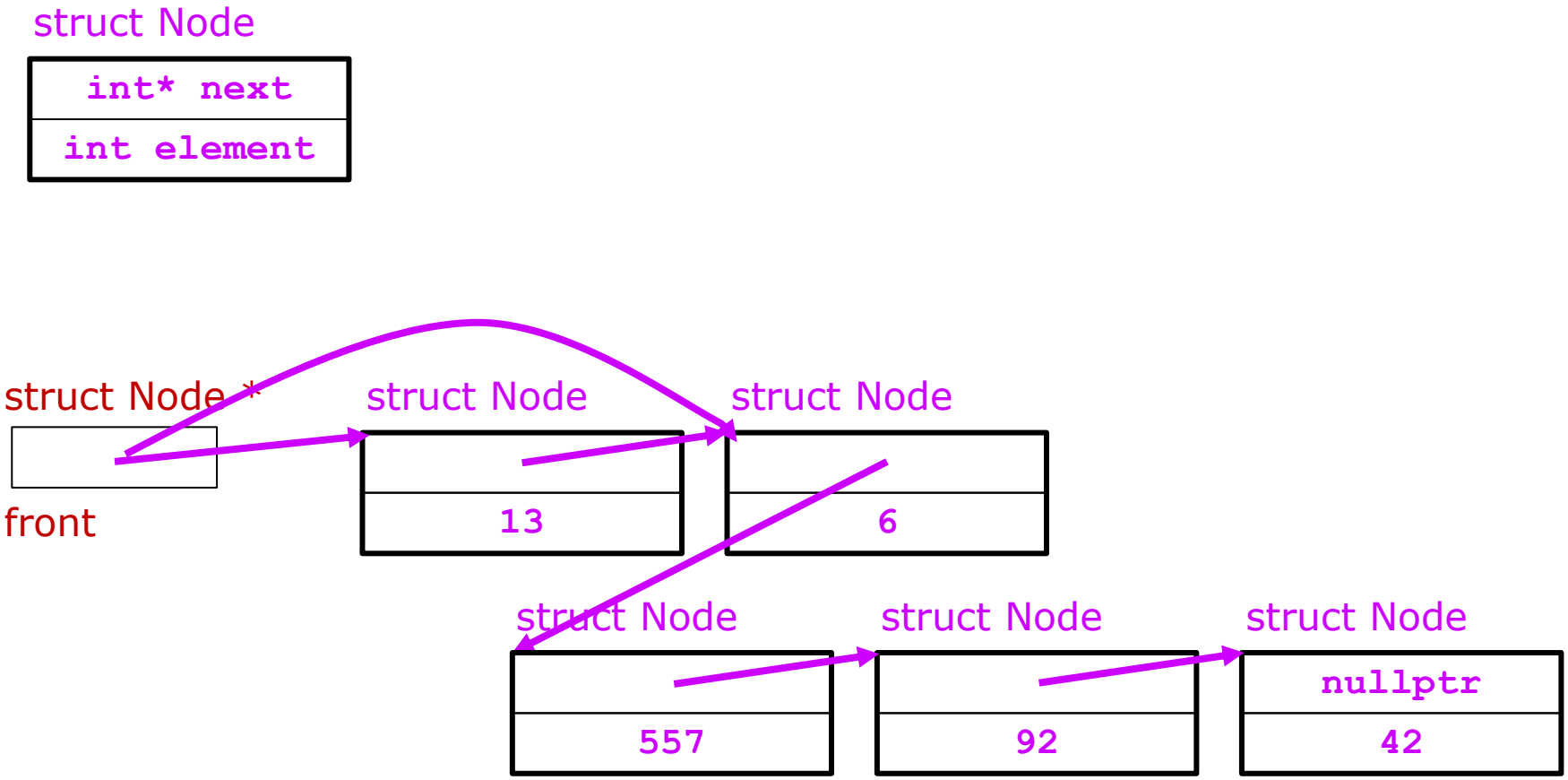
A first look at *linked* lists



A linked list of five elements



Removing the first element



Implementing lists using linked nodes

- **Adding at the end is:**
  - Easy?
  - Hard?
- **Adding at the beginning is:**
  - Easy?
  - Hard?
- **Getting a value by index is:**
  - Easy?
  - Hard?
- **Deleting at the end is:**
  - Easy?
  - Hard?
- **Deleting at the beginning is:**
  - Easy?
  - Hard?
- **Inserting / deleting in the middle is:**
  - Easy?
  - Hard?

Summary

Summary

- **We reviewed dynamic array lists – arrays that grow as needed**
- **We evolved from a struct- to a class-based implementation, focusing on…**
- **…clean interfaces *that hide implementation details***
- **We started exploring *lists* as an Abstract Data Type (independent of implementation details**
- **We started looking at linked, lists a *classic* implementation of lists**
- **We compared the performance tradeoffs between array-based lists and linked lists using pointers**