Lecture 23 — Advanced Arrays

J. Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

August 27, 2016

Acknowledgments: W.D. Bishop

Arrays in Memory

Now that we are familiar with pointers and some basics about memory organization, we can return to the subject of arrays.

It is important to remember the difference between a value type and a pointer type when working with arrays:

- An array of value types provides storage for values;
- An array of pointer types provides storage for pointers.

This has an important implication: A declaration of an array of structure pointers, for example, does not provide storage for fields.

Reference Type Array Example

Consider an array of Student entries where each `Student` object has a name member field and an age member field:

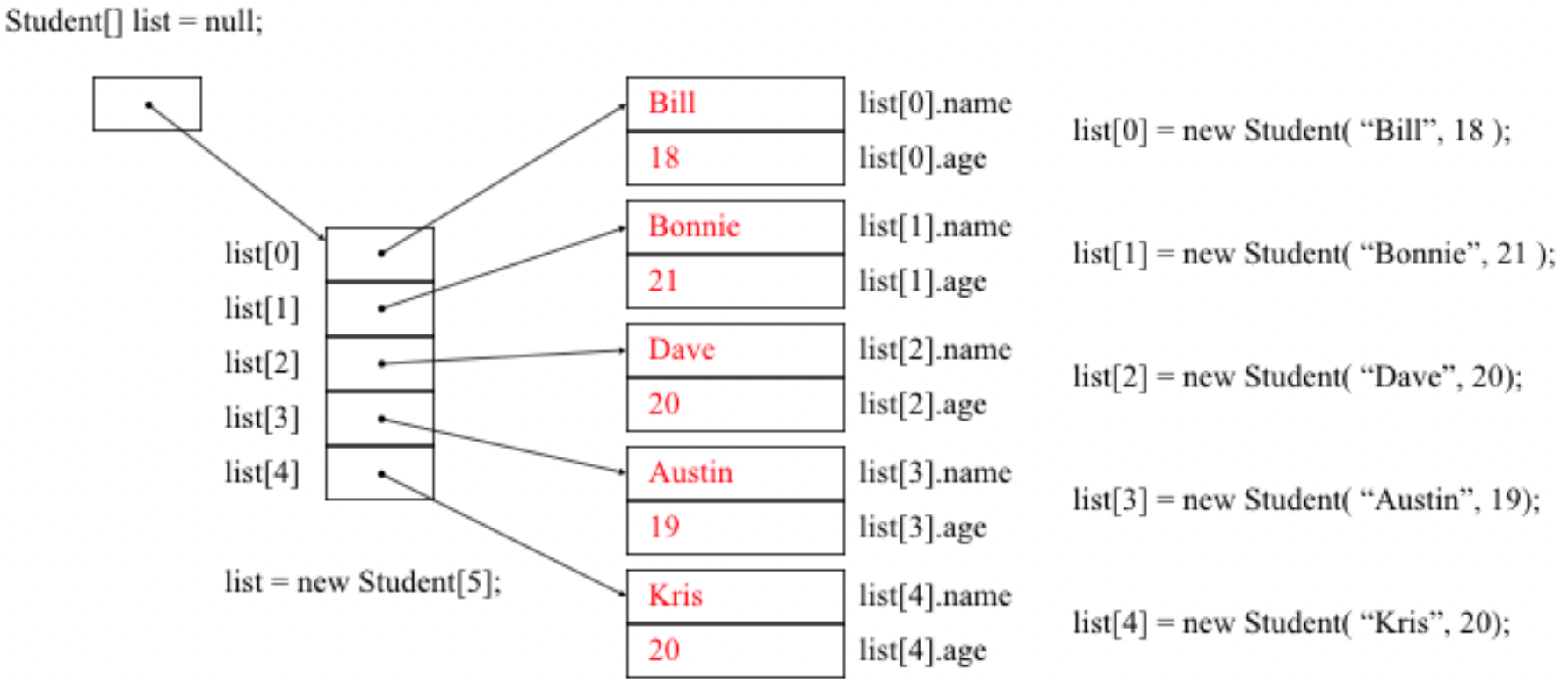
```
struct Student[5] list; // Declare an array of references to student structs

// At this point, list[0], list[1], list[2],
// list[3], and list[4] are null references

list[0].name = "Bill";
list[0].age = 18;
list[1].name = "Bonnie";
list[1].age = 21;
list[2].name = "Dave";
list[2].age = 20;
list[3].name = "Austin";
list[3].age = 19;
list[4].name = "Kris";
list[4].age = 20;
```

Notice that creating the array was not enough.

We also had to populate the structures.

Reference Type Array Example

And here's what that code will produce in memory:



Note that there are six references defined in total.

Regular Arrays

Remember earlier in the term our examination of arrays included multidimensional arrays.

`int[][] jag;`

Declares this in a single array.

Array Storage

Internally, C++ stores arrays using row-major order.

Suppose we ask for an integer array of 3×3: `int[3][3]`.

A 3×3 array of integers stored in contiguous memory locations:

a_{00}	a_{01}	a_{02}
a_{10}	a_{11}	a_{12}
a_{20}	a_{21}	a_{22}

is stored as…

a_{00}	a_{01}	a_{02}	a_{10}	a_{11}	a_{12}	a_{20}	a_{21}	a_{22}

Less Contiguous Array Storage

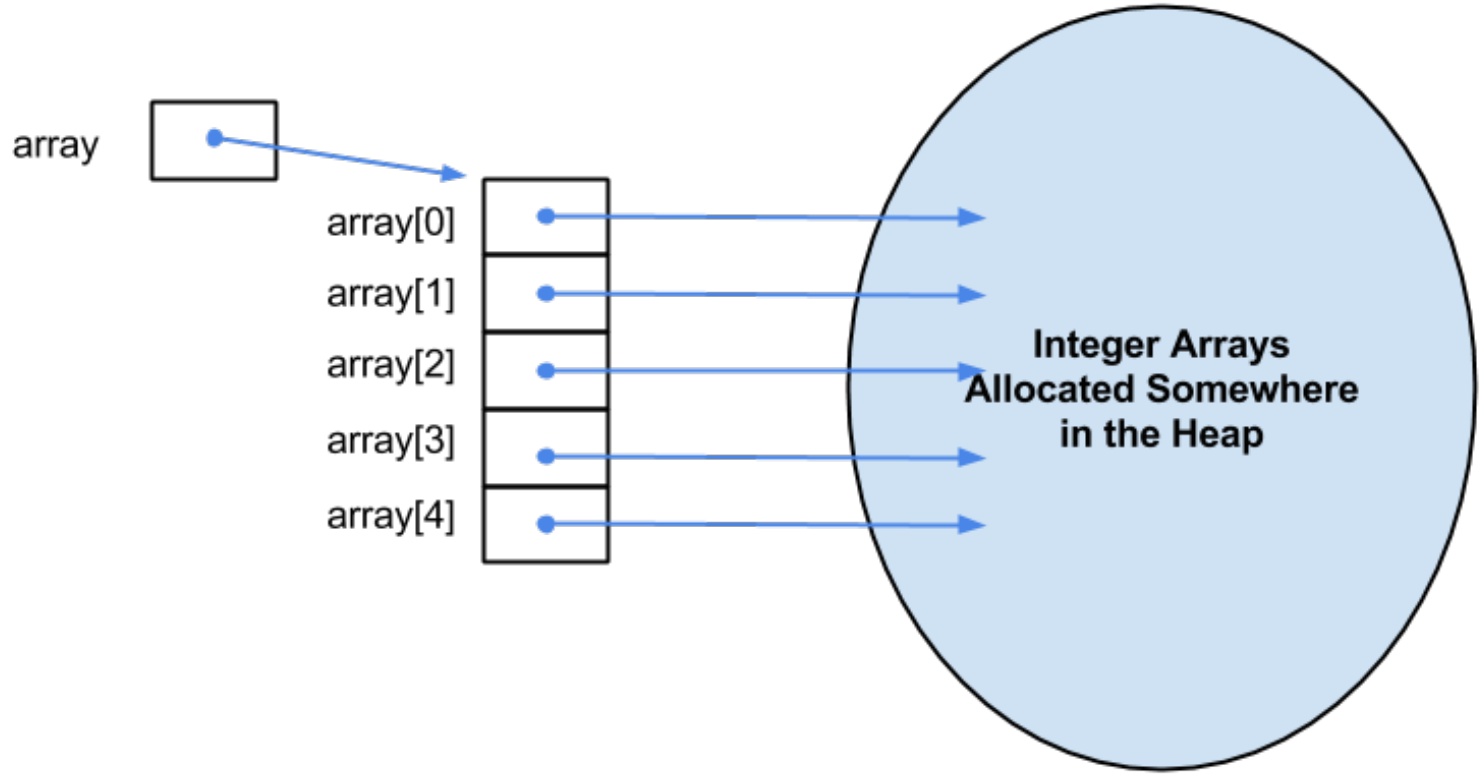
Let's contrast that against an array declared as follows:

```
int n = 5;
int m = 7
int* array = new int[n];

for( int i = 0; i < n; ++i )
{
    array[i] = new int[m];
}
```

What does this look like in memory?

Jagged Array in Memory

Each of the arrays is allocated somewhere on the heap, but it could be anywhere in that relatively large area.



(The first level of the array is also allocated somewhere on the heap).

Jagged Arrays

Arrays declared the first way have nicer syntax and are compact in memory.

Being compact in memory is advantageous because you can do pointer arithmetic to move around within the array.

It is also advantageous because of how CPUs work (which you'll examine in a future course).

Jagged arrays allow more flexibility since each of the arrays need not be the same size.

For a sufficiently large rectangular array, the system may struggle to allocate a single contiguous block of memory.

Memory View of Array Storage

A few notes about the previous diagram:

1. The two-dimensional array is actually stored as a one-dimensional array.
2. The rectangular jagged array is not contiguous but each of the individual arrays are contiguous.
3. If starting address of the array of references is x, the ending address is $x + 9s$ where s is the size of a reference.

Comments on Array Storage

Regular Arrays:

- Implement 1 contiguous storage block that contains all entries of an array.
- May have entries that are value or reference type
- Allow efficient computation of the starting location of any entry using the indices of the array.

Jagged arrays:

- Implement several contiguous storage blocks that contain all entries of an array.
- Define one or more arrays of references that refer to storage locations of the actual entries of an array
- Require more time to access entries due to the additional level(s) of indirection

Enlarging Arrays

An array is of fixed capacity (even if the capacity is user input).

Plan ahead: Allocate an array of size 999 when we aren't sure how many we'll need, and hope that's enough?

What do we do if the array is "full" but we'd like to add more entries?

Reactively: Create a new, bigger array if you need it and copy all the data to the bigger one...?

Enlarging Arrays

The general procedure to enlarge an array is:

1. Request memory for a new array
2. Copy the values over
3. Reassign the original reference
4. Free the memory of the original array

(Shrinking the array follows the same sequence, but it doesn't happen nearly as often as enlarging.)

Enlarging Array Code

First idea – increase the capacity of the array by one.

```
void increase_capacity( int* array, int current_size )
{
    int* largerArray = new int[current_size + 1];
    int* old_array = array;

    for( int i = 0; i < current_size; ++i )
    {
        largerArray[i] = array[i];
    }

    array = largerArray;
    delete [] old_array;
}
```

But this is really inefficient if we have to do it many times.

Enlarging Array Code

A second idea: enlarge the array to twice the original capacity.

```
void increase_capacity( int* array, int current_size )
{
    int* largerArray = new int[current_size * 2];
    int* old_array = array;

    for( int i = 0; i < current_size; ++i )
    {
        largerArray[i] = array[i];
    }

    array = largerArray;
    delete [] old_array;
}
```

Enlarging the array can take a while if the array is large.

Enlarging the array to increase the capacity by one will probably be very inefficient as we'd likely end up enlarging the array many times.

Yet, doubling the capacity when enlarging may result in wasting a lot of memory, such as an array of capacity 200 000 that's only half full...

It would be nice to have a collection of arbitrary capacity...
And can accommodate however many objects we want to add.

About the `delete` Syntax

In the previous examples, we saw `delete [] old_array;`.

The `delete` statement had the additional index brackets.
    This is necessary whenever an array has been allocated with `new`.

If you forget the `[]` you get undefined behaviour: it is not clear what will happen in a given situation.