68. Array of Objects

Objectives

- Review Array and pointer
- Use Arrays to store objects
- Build static sized Array Objects
- Build static sized Array Object Pointers
- Build dynamic sized Array Object
- Build dynamic sized Array Object Pointers

Review

As you know, the size of the array is fixed:

```
const int SIZE = 5;
int x[2 * SIZE];    // OK
int y[42];          // OK
int i = 5;
int z[i];           // WRONG!!!
```

For a “dynamic” array, i.e., an array whose size can be variable, you can use a pointer.

```
int * p = new int[5];          // OK
const int SIZE = 5;
int * q = new int [2 * SIZE];  // OK
int i = 5;
int * r = new int[i];          // OK!!!
```

Memory allocated to a pointer must be de-allocated by the programmer:

```
int i = 5;
int * r = new int[i];
// do something with r[0],...,r[i - 1]
delete [] r;
```

Memory can be allocated to a pointer more than once. The sizes can be different.

```
int * r = new int[5];
// ... do something with r[0],...,r[4]
delete [] r;
...
r = new int;
// ... do something with *r;
delete r;
...
r = new int[10];
// ... do something with r[0],...,r[9]
delete [] r;
```

Like a pointer to a structure variable, if p points to an object with instance variable `x`, then

`(*p).x` is the same as `p->x`

Similarly if the object has method `m()`, then invoking

`(*p).m()` is the same as `p->m()`

Use `->` instead of `*.`

Array of Objects

So far objects are created individually.

What if you want to create an array of objects

```
C obj[10];
```

where `C` is the class?

Try the following example:

```
// Int.h
#include <iostream>
class Int
{
public:
    Int(int x=0)
        : x_(x)
    {
        std::cout << "Int ... x_: " << x_
                  << std::endl;
    }
private:
    int x_;
};
```

```
#include "Int.h"
int main()
{
    Int a[10];
    return 0;
}
```

The Point: Each `a[i] (i=0,...,9)` is constructed using the default constructor. When you declare an array of objects each object in the array calls the **default constructor**.

You **must** have a default constructor if you want to have an array of objects. (Don't forget that if you don't specify any constructor, C++ will give you a default constructor.)

Of course you can still call the constructor one at a time like this if you wish:

```
Int a[3] = {Int(3), Int(-1), Int(42)};
```

Now try to run this (why does it not work):

```
// vec2d.h
#include <iostream>
class vec2d
{
public:
    vec2d(double x, double y)
        : x_(x), y_(y)
    {}
private:
    double x_, y_;
};
```

```
#include "vec2d.h"

int main()
{
    vec2d v[10];
    return 0;
}
```

The point: Each of the `v[0],...,v[9]` calls `vec2d()` (the default constructor) to initialize itself. But there is no default constructor in `vec2d`. There's only one constructor and it must receive two doubles. Remember that! If you want to declare an array of objects without an initializer list, you **MUST** have the default constructor used to construct the objects in the array.

Exercise. Fix the following class so that an array of `WeatherControl` objects can be created with all instance variables initialized to 10.

```
// WeatherCtrl.h
#ifndef WEATHERCTRL_H
#define WEATHERCTRL_H

class WeatherCtrl
{
public:
    WeatherCtrl(double temp, double pressure)
        : temp_(temp), pressure_(pressure)
    {}
    double get_temp() const { return temp_; }
    double get_pressure() const { return pressure_; }
    void set_temp(double);
    void set_pressure(double);
private:
    double temp_;
    double pressure_;
};

#endif
```

Suppose we have the following class:

```
class Int
{
public:
    Int(int x)
        : x_(x)
    {}
private:
    int x_;
};
```

Now suppose you want to create an array of `Int` objects with

```
Int a[10]; // want a[i].x = i
```

First Method:

```
class Int
{
public:
    Int (int x = 0)
        : x_(x) {}
    void set(int a) {x_ = a;}
private:
    int x_;
};

int main()
{
    Int a[10];
    for (int i = 0; i < 10; i++)
        a[i].set(i);
    ...
}
```

1. Forced to have a default constructor.

2. Construct each `a[i]` with default constructor.

Important point: **Two** methods called for each object to set the initial values.

3. set each object

So note that in the above each `a[i].x` was first set to `0` by the constructor ... then `a[i].x` was set to `i`.

The Point: `a[i].x` was set two values.

This is no big deal for the `Int` class because not a lot of work was done in the constructor. But... what if a certain class `C` does a lot of work in the constructor? Example: `C` has `1000000` member variables and the default constructor sets all to `0` and then you need to set them all to `i`?

Second method:

```
int main()
{
    Int * a[10];
    for (int i = 0; i < 10; ++i)
        a[i] = new Int(i);
    // do something with *(a[i])
    ...
}
```

1. Declare array of **pointers.**

2. Call constructor to construct object in the heap and return pointer.

The point: Each `a[i]` is not an object, but a pointer to an object. We can then construct the object in the heap with whatever constructor call that we choose and then point `a[i]` to the object.

The point: There is no constructor call when you do

```
Int * a[10];
```

or when you do

```
Int * p;
```

The point: There is no "default constructor call" for pointers. Pointers are not objects.

Let's compare the two methods.

In the first method, when we create an array of objects the default constructor is called for each object.

In the second method, for the array of pointers to objects, you can call a constructor for each object in the heap that each pointer of the array points to.

How do you deallocate memory correctly for the second method?

```
Int * a[10];
for (int i = 0; i < 10; ++i)
    a[i] = new Int(i);
// do something with *(a[i])
for (int i = 0; i < 10; ++i)
    delete a[i];
```

Compare the above with this:

```
Int * a = new Int[10];
for (int i = 0; i < 10; ++i)
    a[i].set(i);
delete [] a;
```

Make sure you see the difference between

- Array of Pointers
- Pointer to an Array

Note that for the second method:

```
int main()
{
    Int * a[10];
    for (int i = 0; i < 10; ++i)
        a[i] = new Int(i);
    // do something with *(a[i])
}
```

The size of the array is fixed. To have a “dynamic” array where the size can change, you can use a pointer.

```
int size;
std::cin >> size; // enter integer > 0
Int ** a = new Int*[size];
for (int i = 0; i < 10; ++i)
    a[i] = new Int(i);
...
// Exercise on pointers to objects:
// Assign 43 to x of Int object a[0]
// is point to.
```

The above is analogous to this:

```
int size;
std::cin >> size; // enter integer > 0
Int ** a = new Int*[size];
for (int i = 0; i < 10; ++i)
    a[i] = new Int(i);
// ... do something with *a[i]
for (int i = 0; i < size; ++i)
    delete a[i];
```

Note:

```
Int ** a = new Int*[size];
Int ** b = new (Int*)[size];  //WRONG!!!
```

Comparison:

```
Int a[10];   //Fixed size, default constructor
Int * b[10]; //Fixed, can call constructor
Int ** c = new Int*[size]; // variable size and can
                           // call constructor.
```

What if you want to do this:

```
Date a[10]; // all 10 initialized to 12/25/2009
Date b[20]; // all 20 initialized to 12/25/2010
```

Keep default year, month, day values in Date class for initialization purposes. Allow defaults to be changed. But default values for the constructor is fixed.
See notes on Static for details. Such values are more associated with the class than with individual objects.

Exercise. Write a program that does this:

- Prompt the user for s.
- Create a dynamic array of size s of Int objects initializing them so that you get a Fibonacci sequence.
- For instance if you enter 7 the integers modeled in the array are:
  1, 1, 2, 3, 5, 8, 13

You can also have 2-dim array of objects.

Example:

```
Int a[5][6]; // default constructor called on each
             // a[i][j]
for (int i = 0; i < 5; i++)
    for (int j = 0; j < 6; j++)
        a[i][j].set(42);
```

In this example, the array has fixed sizes, and you loose control over the constructor call.

Use an array of object pointers to control constructor call

```
Int * a[5][6]; // 5-by-6 array of Int *
for (int i = 0; i < 5; i++)
    for (int j = 0; j < 6; j++)
        a[i][j] = new Int(42);

// do something useful with *(a[i][j])

for (int i = 0; i < 5; i++)
    for (int j = 0; j < 6; j++)
        delete a[i][j];
```

The above is an array of **fixed** sizes.

So... what if you want the sizes to be dynamic?

```
int rows, columns;
// ... prompt user for rows columns
Int * a[rows][columns]; // WRONG!!!
```

The following is a 2-dimensional dynamic array of objects, i.e., the row and column size can be variables. However, the value in the array are objects (not pointers yet … we'll get to that in a bit...)

```
int rows = 5, columns = 6;
Int ** a = new Int*[rows]; // a[0], ..., a[rows-1]
                           // are all Int * values
for (int i = 0; i < rows; ++i)
{
    a[i] = new Int[columns]; // a[i][j] for j = 0,...,
                             // columns-1 are Int
    for (int j = 0; j < columns; ++j)
        a[i][j] = 42;
}
```

```
int rows = 5, columns = 6;
Int ** a = new Int*[rows]; // a[0], ..., a[rows-1]
                           // are all Int * values
for (int i = 0; i < rows; ++i)
{
    a[i] = new Int[columns]; // a[i][j] for j = 0,...,
                             // columns-1 are Int
    for (int j = 0; j < columns; ++j)
        a[i][j] = 42;
}
// do something with a[i][j]
// De-allocate memory used by a[0], …, a[rows-1] and a
for (int i = 0; i < rows; i++)
    delete [] a[i];
delete [] a;
```

Exercise. Write a program that does this:

- prompts the user for the row and column size of a 2-dim array of integer values. (Just the regular int).
- Fill the 2-dim array with random integers.
- Print the values in the array in a 2-dim grid.
- Compute the row, column, and overall average of all the values (up to 1 decimal place).
- Deallocate Memory used.
- Here is what a 3 by 3 case looks like

```
 0   3   3 |2.0
 2   9   0 |3.6
 8   2   6 |5.3
-----------+
3.3 4.6 3.0 3.6
```

Exercise. Run this program. Explain why it does not work.

```
int floors = 3, rooms_per_floor = 10;
WeatherControl ** wc = new WeatherControl*[floors];
for (int floor = 0; floor < floors; ++floor)
{
    wc[floor] = new WeatherControl[rooms_per_floor];
}
delete [] wc;
for (int floor = 0; floor < floors; ++floor)
{
    delete [] wc[floor];
}
```

From previous example, each `a[i][j]` is an `Int` object. You lose control over calling constructor.

Now we want the row and column sizes to be dynamic **and** the values in the array to be pointers so that we can call the constructor for each

pointer…

```
int rows = 5, columns = 6;
Int *** a = new Int**[rows];
for (int i = 0; i < rows; i++)
{
    a[i] = new Int*[columns];
    for (int j = 0; j < columns; j++)
        a[i][j] = new Int(42);
}

// do something with (*a)[i][j]
for (int i = 0; i < rows; i++)
{
    for (int j = 0; j < columns; j++)
        delete a[i][j];
    delete [] a[i];
}
delete [] a;
```

Good reasons for building a class for handling these initialization and cleanup operations!!!

Here a summary:

- For a static sized 2-dim array of `Int` objects:

```
Int a[3][5]
```

- For a static sized 2-dim array of `Int` pointers:

```
Int * a[3][5]
```

- For a dynamic sized 2-dim array of `Int` objects:

```
Int ** a
```

- For a dynamic sized 2-dim array of `Int` pointers:

```
Int *** a
```

The point:

```
Int *** a
```

To make the first dimension dynamic.

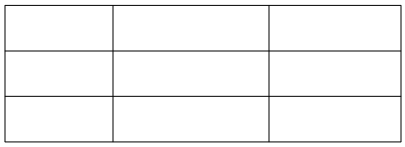
To make the second dimension dynamic.

To make the value in the 2-d array dynamic

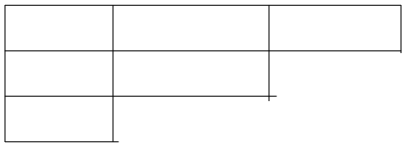
Exercise.

- Create a 2-dim (dynamic) array of `Int` object pointers. The size of the first and second dim is randomly chosen from integer 1,2,...,10.
- Initialize the pointers by allocating memory for `Int` objects and initializing the `Int` objects with integers randomly chosen from 1,2,...,10
- Print all the integer values "in" this array in a 2-dim grid.

All the arrays up to this point have been rectangular, i.e., the arrays as a container values look like this.



Create an array that is triangular, i.e. for row 0
there are 3 columns, for row 1 there are 2 columns, etc. (They are called jagged arrays.)



For the static size 3-dim array of `Int` objects:

```
Int a[3][6][4]
```

For the static size 3-dim array of `Int` pointers:

```
Int * a[3][6][4]
```

For the dynamic size 3-dim array of `Int` objects:

```
Int *** a
```

For the dynamic size 3-dim array of `Int` pointers:

```
Int **** a
```

Phew!!!!

Exercise. In a function `f(a, b, c)` create a dynamic array of `Int` pointers with sizes `a, b, c`. Initialize all the pointers by allocating the memory and initializing the objects the pointer points to with a random integer. Make sure you de-allocate memory used.

In your `main()`, write a `for` loop that calls `f(10, 10, 10)` and print a message for each iteration of the loop. If the printing of the messages slows down, it tells you that you probably have a memory leak.

Exercise. Write a class that models a dynamic 2-dim array of integers. The following usage will tell you how it works:

```
IntDynArr a(5,6); // 5-by-6 int array
a(1, 2) = 42; // set "a[1][2]" to 42
std::cout << a(1, 2) << std::endl;
```

Hint: `a(1, 2)` is `a.operator()(1, 2)`. You need to define `int & operator(int, int)`.

Exercise. Rewrite your class so you can execute this:

```
IntDynArr a(5, 6);
a[1][2] = 42;
```

Hint: Note that

```
a[1][2]
```

Is not `a.operator[][](1,2)` because there is no `[][]` operator!!!
Instead `a[1][2]` is `a.operator[](1).operator[](2)`

Summary

The objects in an array are initialized using the default constructor.

If you want to get away from using the default constructor (i.e., you want to explicitly call the constructor) but still want a static sized array, you can use an array of pointers.

If you want to get away from using the default constructor (i.e., you want to explicitly call the constructor) and you want to get away from a static sized array, you can use a pointer to an array of pointers.