Memory/Arrays

Random Numbers:
#include <stdlib.h>
...
int x = rand();  // generates a random number between 0 and a max number defined in stdlib

Problem: if the random generator is not given a specific seed, it will always start with the same seed. So you will get the same sequence of random numbers.

We want to give it a seed, and each time we run our program, we want a new seed. So we often use the current time as our new seed.

To generate a seed using the current time:
#include <time.h>
...
srand(time(NULL)); // creates a seed based on the current time (down to the millisecond)
// Only need to create a seed once in a program
// Must create seed BEFORE you use rand for the first time.
int x = rand(); // now rand uses this seed in the calculation of the random number.
// Now you won’t always get the same sequence of random numbers

Question: How would you generate a random number between 1 and 10?

Some Math Stuff (in the Math Library):
- #include <math.h>
- ...
- double d = 200.374;
- int i = 100;
- int j = -100;
- cout << sin(d) << endl; // -0.634939
- cout << sqrt(i) << endl; // 10
- cout << pow(i, 2) << endl; // 10000
- cout << floor(d) << endl; // 200
- cout << abs(j) << endl; // 100 - only works with ints
- cout << fabs(d) << endl; // 200.374 - only works with doubles and floats

/***************************************************************************/

Arrays
*******************************************************************************/

Creating Arrays:
The following creates an array of 5 ints is filled with 3,2,4,1,7

    int a[5] = {3,2,4,1,7};  // note squiggly brackets

The following creates an array of 3 floats is filled with 3.2,1.4,0;

    float b[3] = {3.2,1.4};

The following creates an array of spaces for 7 ints (but the array holds no values yet);

    char c[7] = {7};
int c[7];
If you do the following:

c[3] = 24;
the array at 3 holds 24, and the rest holds nothing (if you print, will print out whatever happens to be leftover there – it will look like gibberish or some really strange numbers most likely)
cout << c[2] << endl; // might print -328918973

The following makes an array of 3 ints, holding 4,1,2

int d[] = {4,1,2};

We can do this because when this line is executed, the compiler knows how much space to set aside in memory for the array. The compiler knows to set aside space for THREE SEQUENTIAL INTEGERS in memory.

*d holds the address of the space for the first integer. (GOT THAT???)*

THIS IS CRITICAL TO UNDERSTANDING ARRAYS!!!  *d is a variable that holds an address (just like a pointer!)  It holds the address of the space in memory where the first value in the array is or will be located.  d is an address.  d is an address of the first value in an array.  d is an address.  d is an address.  Have I emphasized this enough yet?*

So this is valid in c++

Next:

int d[];
d[0] = 4;

**CANNOT DO THIS!!!!!!!!!!**

d[] is indicating that we need an address for an array of ints in memory. But we haven’t actually made an array of ints. We don’t know how many spaces to set aside, and equally, we don’t know what the address of the first value in the array should be.

*Can we set aside space? Hold that thought…*

**Think about it:**

Given:

int arr[4] = {2,16,8,23};

If we know the size (number of bits) of ints, and that arrays occur sequentially in memory, what is the absolute minimum we need to know about the array to find any of the values in the array?

**The only thing you need to know about an array to access all the values in the array is the ADDRESS OF THE FIRST VALUE OF THE ARRAY IN MEMORY!!!  Because the values in an array are sequential in memory (next to each other), so if you have the address of the first value, you add to that address 1*sizeof(int) to get to the next value, you’d add 2*sizeof(int) to get to the 3rd value, etc.  You just need to know the type and the address of the very first value in the array to access every value in the array.**

**Passing arrays into a function:**

Given the following array:

```c
int x[4000000];
for (int i = 0; i < 4000000; i++) {
    x[i] = i;
}
```
In terms of memory space, do we want to make a new, local copy of the array when we call a function?

No, this wouldn’t make sense, especially for a language that prides itself on creating small, extremely efficient executable code!

Instead, the default for arrays is to pass the address of the first value of the array into a function.

There are 2 ways to get the address of the array (aka a pointer to the array):

```
int arr[4] = {3,2,1,4};
ocout << &arr[0] << endl; //method 1: address of first value in array

cout << arr << endl; //method 2: same value, address of array, of where first value in array is
```

**Try on Computer:**

1) Create an array of integers.
2) Print out the address of the array using both methods above
3) Now print out the address of each of the values in the array using method 1 (&arr[1], &arr[2], etc.)
   See how they’re sequential?

**Arrays and Functions:**

*Note:* There’s no straightforward way of getting size of an array, especially after you’ve passed it into a function. Your best bet is to pass the size of the array into the function as a separate parameter.

To pass an array into a function you can do either:

```
int main() {
    int x[4] = {3,2,4,1};
double k = getAverage(&x[0],4); // gets the address of the first value in the array
    // Or
    double l = getAverage(x,4); // the address of the first value in the array
    cout << k << endl;
}
```

Then the function definition would be:

```
double getAverage(int *arr, int size) { // a pointer, a variable that holds an address
    //note that the *arr looks EXACTLY like call by pointer with an individual int.
    //There’s no way to tell whether *arr is an address of an int or an array of ints.
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i]; // now, in essence, the value at (the address of arr + i)
    } //for
    double avg = double(sum) / size;
    return avg;
} //getAverage
```

**Alternative:**

```
double getAverage(int arr[], int size){ //also holds address of the first value in the array
    This way is often preferred because it makes it clear that you’re passing in an array
}
```

**Try on Computer:**

Try the following on your computer. Note what happens. Can you figure out why (hint: addresses)?
```c
int main() {
    int arr[7] = {10,20,30,40,50,60,70};
    func(arr,7);
    func(&arr[0],7);
    func(&arr[2],5);
    return 0;
}

void func(int *x, int size) {
    for (int i = 0; i < size; i++) {
        cout << x[i] << " , ";
    }
    cout << endl;
    return;
}
```

**Dynamically allocated arrays**

What if we don’t know the size of the array when we write the code?

- Maybe the array size is random
- Maybe we’re reading in data from a file

What if we want to create an array inside of a function and then have the array continue to exist after the function is done running? (So it’s not a local variable)?

We can “**Dynamically allocate**” the array

- Short version: create a variable that will hold the address of the first value of an array (at some point)
- Then, when we want to, we set aside space in memory for the array and place the address of the first value of that space into our variable.

**Memory Management (Stack, Heap)**

There are basically two types of memory: the stack and the heap.

The stack is out of our control. The compiler puts things into the stack memory and it also removes things from the stack memory. We cannot explicitly place things in the stack, nor can we explicitly remove things from the stack. Again, the compiler controls what goes into and comes out of the stack and the timing of when this happens.

E.g.,

```c
int main() {
    func(3);
    cout << k << endl;  //what is printed here? Why?
    return 0;
}

void func(int x) {
    int k = pow(k,x);
    return;
}
```

Heap (this is new to C++): We control!!

We purposely and explicitly put things in the heap part of memory.

We need to keep track of the address of where we put variables in the heap! (or we’ll lose that variable and it will become garbage)
Variables we created on the heap continue to exist in that heap part of memory until we explicitly get rid of them.

How do we put things on the heap? **New**

```cpp
new

    new int; // sets aside memory on the heap for an int (but no variable points to it
    // this is useless on its own!

    int *x; // just sets aside space for an address that will hold an int - just the space for
    // the address!
    x = new int; // now x points to (holds the address of) an int in memory (on the heap)
    *x = 34; // now the x points to 34; meaning that at the address x holds, there's a 34
```

**NOTE:** you can combine into one step:

```cpp
int *x = new int;
```

What if there's not enough space on the heap?

```cpp
int *x;
if( !(x = new int)) {
    cout << "Error: out of memory." << endl; // checking to see if there's space on the heap
    exit(1);
} // if
else {
    *x = 2;
} // else
```

Now x will continue to exist on the heap in memory until you explicitly remove it from the heap.

How do you remove it from the heap? **Delete**

**Delete**

When you're done with a variable, you want to remove it from the heap.

Delete: frees up the space on the heap:

```cpp
int *x;
if( !(x = new int)) {
    cout << "Error: out of memory." << endl; // checks if there's space on the heap
    exit(1);
} // if
else {
    *x = 2;
} // else
    delete x; // deallocates the space in memory
```

What about arrays? How do we create an array on the heap? And how do we free up the memory space once we're done using it?

```cpp
int *x = NULL; // Pointer initialized with null (default is undefined)
cout << "Enter the number of numbers you want" << endl;
int y;
cin >> y;
x = new int[y]; // Request memory on heap for space for y ints (sequentially)
for (int k = 0; k < y; k++) {
    x[k] = k;
} // for
```

And now to free an array?
delete [] x;

Try on paper:

Why can’t you just do:

delete x;

to delete the above array? It compiles...

What if you create an array inside of a function and want it to exist outside of the function? Well, clearly it has to be created in a manner that places it on the heap. Otherwise it will be local, and, because the compiler controls local variables, it will disappear when the function is done running. If the array is created on the heap, its address on the heap must be returned from the function so that code outside of the function can find the array. To do that, you’d do the following:

```c++
int * createArray(int size);
int main() {
    int *a; //a holds an address that points to an int (or the first in a list of ints)
    a = createArray(4);
    for (int x = 0; x < 4; x++){
        cout << a[x] << endl;
    } //for
    delete[] a;
    return 0;
} //main
int *createArray(int size) { // what is returned??
    int *r = new int[size]; // This array had to be put on the heap.
    r[0] = 3;
    r[1]= 2;
    r[2] = 4;
    r[3] = 1;
    return r; // or return &r[0];
} //createArray
```

Note the lines:

```c++
int *a;
    a = createArray(4);
```

a is a variable that holds an address. The function createArray() returns an address. so outside of the function createArray, a will hold the address of the array we placed on the heap.

Stack Versus Heap:

<table>
<thead>
<tr>
<th>Stack (Memory)</th>
<th>Heap (memory in a different location)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Temporary memory storage for variables associated with each function</td>
<td>- not managed automatically</td>
</tr>
<tr>
<td>- When a function is called – its variables are pushed onto the top of the stack</td>
<td>- Larger than the stack</td>
</tr>
<tr>
<td>- When the function ends – all of its variables are popped from the top of the stack</td>
<td>- Variables in heap can be accessed globally (if we keep track of their address on the heap)</td>
</tr>
<tr>
<td>- Stack grows and shrinks as function variables are pushed and popped</td>
<td>- Slower to access</td>
</tr>
<tr>
<td>- No need (or ability) to manage memory</td>
<td>- Less efficient than the stack</td>
</tr>
<tr>
<td>- Stack has size limits</td>
<td>- Manually allocate space for variables on the heap and then free them when their use is done</td>
</tr>
<tr>
<td>- Stack variables only exist while the function created is pushed on the stack</td>
<td>- Nice and fast</td>
</tr>
<tr>
<td>- We do not control what goes onto or comes off of the stack.</td>
<td>- We do not control what goes onto or comes off of the stack.</td>
</tr>
</tbody>
</table>
1) What does this give you?

```c
int *func1(int x);
int main()
{
    int *ar2 = func1(4);
    cout << ar2 << endl;
    for (int k = 0; k < 4; k++) {
        cout << ar2[k] << ", ";
    }
    cout << endl;
    return 0;
} //main
int *func1(int x) {
    int arr[x];
    for (int i = 0; i < x; i++) {
        arr[i] = pow(i, 2);
    }
    return arr;
} //func1
```

2) What does this give you?

```c
void arrfunc1(int arr[], int &num);
int main()
{
    int a[] = {12, 2, 3, 18, 20, 4, 7, 22, 3};
    //indices: 0 1 2 3 4 5 6 7 8
    int tot = 0;
    arrfunc1(&a[3], tot);
    cout << tot << endl;
    return 0;
} //main
void arrfunc1(int *arrx, int &num) {
    for (int i = 0; i < 3; i++) {
        num += arrx[i];
    }
} //arrfunc1
```

3) CHALLENGING! What does this give you?

```c
void arrfunc1(char arr[], char arr2[]);
char *arrfunc2(char arr[], int &len);
int main()
{
    char a[] = {'k', 'a', 'p', 'd', 'e', 'b', 'm', 'a', 'c', 'u', 'g', 'g', 'h', 'p', 's', 'i', 'n', 'g'};
    int len = 18;
    char *b = arrfunc2(a, len);
    for (int i = 0; i < len; i++) {
        cout << b[i] << " ";
    }
    cout << endl;
    return 0;
} //main
```
void arrfunc1(char *arrx, char arry[]) {
    for (int i = 0; i < 3; i++) {
        arry[i] = arrx[i];
    }
}

char *arrfunc2(char arr[], int &len) {
    int newlen = len/2;
    char *arr2 = new char[newlen];
    for (int i = 3; i <= len - 3; i += 6) {
        arrfunc1(&arr[i], &arr2[(i-3)/2]);
    }
    len = newlen;
    return arr2;
}

*X:*
X is a pointer. X holds the address of something else. Thus it must be made to point to:

A: SOMETHING THAT EXISTS (you’ve got to get the address of it):

    int y = 3;
    int *x;
    x = &y; // x now points to y - holds the address of y

Or B: SOMETHING YOU CREATE (using new):

    int *x;
    x = new int[4]; //x now holds the memory location of space for 4 ints, technically the address of the first space in the series of 4 consecutive ints (this address is in the heap)

x is a pointer, or address holder. Somehow, somewhere, the address must come to be before x can hold that address

new always returns an address in memory!

And if something is new-ly created, it should be delete-d (AKA Garbage collection). Anything on the heap should be deleted!

What about multi-dimensional arrays?
You must first allocate an array of addresses,

And then for each address in the array, you must allocate an array of ints (or whatever type you want!)

```
array of addresses

X ------- 0 1 2
         1 2 3
         2 3 4
         3 4 5
```
int **x = NULL;  // Pointer initialized with null
x = new int *[4];  // Allocate memory on heap for a 4x3 array
for (int i = 0; i < 4; i++) {
    x[i] = new int[3];
} //for
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 3; j++) {
        x[i][j] = i+j;
    } //for
} //for
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 3; j++) {
        cout << x[i][j];
    } //for
    cout << endl;
} //for

Deleting Multidimensional Arrays:
int **x = NULL;  // Pointer initialized with null
x = new int *[4];  // Allocate memory on heap for a 4x3 array
for (int i = 0; i < 4; i++) {
    x[i] = new int[3];
} //for
for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 3; j++) {
        x[i][j] = i+j;
    } //for
} //for
Try on paper:

Why can't we just do this?

delete [] x;  // compiles just fine...

To delete a 2-dimensional array:
int **x = NULL;  // Pointer initialized with null
x = new int *[4];  // Allocate memory on heap for a 4x3 array
for (int i = 0; i < 4; i++) {
    x[i] = new int[3];
} //for
for (int i = 0; i < 4; i++) {
    delete [] x[i];
} //for
delete [] x;

Try on paper:

What is printed out?

int x = 3;
int y = x;
int *z = &x;
cout << “x is “ << x << endl;
cout << " y is " << y << endl;
cout << " z is " << *z << endl;
x += 4;
cout << " x is " << x << endl;
cout << " y is " << y << endl;
cout << " z is " << *z << endl;

Can I do this?

z = 3;

Lab 2 Part b:
(86 pts, due Thurs, Sep 12)

1. (6 pts) Write a function that takes as input parameters a length parameter (an int), and an int parameter that will be modified using pass by reference. When the function is called the second parameter is initialized to -1 (before the function call). The function should generate a random array the length of the length parameter, with the numbers between 1 and 50. The function should print the array, and then locate the smallest value in the array, modifying the third parameter to be the smallest value. Make sure you print this value after you’ve returned from the function.
   
   Note: to print the array in a way that is nicely readable, you’d use a tab or a “,” instead of an endl
   
   So your line of code would look something like this:
   
   cout << arr[x] << “, “;
   
   And then when you’re done printing out the array (after the loop), you’d then print out the endl
   
   cout << endl;

2. (6 pts) Write a function that takes an input parameter an int and returns nothing. It then generates an array of random numbers the length of the int parameter. It fills the array with random numbers between 0 and 50. It then prints out the array.
   
   Without creating a new array, the function then reverses the array and prints out the reversed array.
   
   So if the first array was:
   
   22,10,8,5,3,18
   
   The reversed array would be:
   
   18,3,5,8,10,22
   
   Note: the numbers in the array must be reversed. Don’t just print out the array backwards.

3. (10 pts) Write a function that takes as an input parameter an integer that will represent the length of the array and a second integer that represents the range of numbers the random number should generate (in other words, if the number is 50, the random number generator should generate numbers between 0 and 49 including 49.
   
   The function then sorts the list by traversing the list, locating the smallest number, and printing out that smallest number, then replacing that number in the array with the second parameter +1 (so in our above example, it would be 51.) Continue to do this until every number in the original array is printed out in order.

4. (3 pts) Write a function that creates a new variable on the stack. Give the variable a value of 3. Print out the value and the address of the variable within the function. Return the address of this variable, and make sure that main has a pointer set to the returned variable (int *x = func();)
   
   Print out the address of the variable and the value in the variable in main. Did this work? Did it compile?
   
   Include a comment on those 2 questions.

5. (2pts) Write a function that takes an array of integers as an input parameter (the address of the first value of the array) and an integer for the size of the array. It returns nothing. It prints out the array as a single line, with commas between each number, and when the array is finished being printed, it prints an endl; so that we flush the buffer and move to a new line. (I’m having you write this function because you’ll be wanting to print out arrays a lot in the following exercises).
   
   Note: you’ll be using this function after running other functions to check out your results.

5b. (2 pts) In main, write a loop that generates an array of random integers on the stack. The length should be 25, and the integers should be between 1 and 10. Use this array to test your function that prints out an array.
Note: there are a few functions below that require as an input parameter an array of integers and the size of the array. This is how the arrays are created, although the length and range may differ.

6. (5 pts) Write a function that takes as input parameters (using call by pointer) 3 integers. It generates a random number between 25 and 50 (not including 50). It then creates an array on the memory heap of that length. It generates a random high number between 5 and 10 and a random low number between -5 and -10 and fills in the array iteratively with random numbers between the high and the low numbers*, and it returns that array. The input parameters should be modified so that it holds the length of the array, the high value, and the low value. In main, call the function 5 to print out the array.

*not including the high – in general when we specify a range, we include the first value but not the last. If I forget to say that in the future, you can assume that’s what I intended.

7. (4 pts) Write a function that is almost exactly the same as the function above, only it takes an input parameter an integer (pass in a number between 25 and 50). Inside the function create an array on the stack instead of the heap. Fill it with random numbers as above. Return the address of the first value of the array, and then in the main use function 1 to print it out. This should NOT work. In comments explain why. (Note that you should comment this function out before turning your code in.)

8. (3 pts) Write a function that takes as an input parameter an array of integers (and the size of the array). The function should print out the address of every value in the array.

9. (3 pts) Write a function that takes as an input parameter an array of doubles. The function should print out the address of every value in the array. See how this works?

10. (5) Write a function that takes as an input parameter an array of integers and the size of the array as a pointer. This function should go through the array and create a new array on the heap that removes all double-numbers that occur next to each other. The function should modify the arraysize parameter to the length of the new array with the multiple occurrences of numbers next to each other removed, and it should return the newly-created array. In other words, if the input array is:
    
    5, 4, 4, 3, 6, 6, 6, 8, 9, 5, 1, 3, 8, 8, 1, 8, 9, 9, 3, 2, 2, 6, 1

    The returned array would be:

    5, 4, 3, 6, 8, 9, 5, 1, 3, 8, 1, 8, 9, 3, 2, 2, 6, 1, 2

    And the size would be adjusted to: 18

HANNING WINDOW:

For this next part, you will be writing a low pass filter using a very basic hanning window. The idea behind a low pass filter is to get rid of high frequencies, or, in essence, smoothing out the massive outliers in an array. The way to do this is to take a window size, usually an odd number in length, in which you take the average of the values before and after a value in an array and replace that value with the average. So, for instance, if you had an array as follows:

3, 8, 2, 5, 1, 4, 6, 0, 2

And you had a window size of 3, you would go through the array in window sizes of 3 and replace each center value with the average. For those values at the beginning and the end, in which you can’t have a window on both sides, you’d replace the values with a 0. Otherwise each value gets replaced with its window average.

The resulting array would be:

0 4 5 2 3 3 2 0

Which is a much smoother array with fewer outliers.

Now, an even better way of smoothing out the outliers is to weight the window. So, in the process of averaging, the values farthest from the center value in the window are weighted the lowest, whereas the values closest to the center of the window are weighted the highest. This is known as a Hanning window.

For instance, if you have a window size of 5, you might weight the values by multiplying the values at location 1 and 5 with 1, the values at location 2 and 4 with 2 and the values at location 3 (the center) with 3, and then dividing the sum by 9 to get the weighted average. So in this case, given the following array and a window size of 5,

3, 8, 2, 5, 1, 4, 6, 0, 2

HANNING WINDOW:
Using a hanning window for a filter, you’d get:

0, 0, 4, 3, 3, 3, 3, 0, 0,

(Note that this is even smoother than the really simple filter we used above).

So you can see where this is going:

11. (5 pts) Write a function for the hanning window. For this function, you should take as input parameters a part of the array (remember – whenever you pass in an array, no matter what you pass in, it is always interpreted as the address of the first value of the array. So if I passed in &arr[3], when that address enters the function, it will assume that the 4th address is the first address in the array. So DO NOT make a copy of each window and send that into the function – that is just wasteful and makes no sense). Assume the window size is an odd number in size. Weight the values appropriately. Return the weighted and averaged value.

12. (10 pts) Write a function for filtering the array. This function should return a new array. You can’t write over the old array (why not?). You will have to create a new array on the heap, and return that new array. This function should take as input parameters the original array and the size of the original array, and should create a new array that has been filtered using the hanning window function. Print out both arrays to see and check the difference.

13. (12 pts) Write a function that takes as input parameters an array, the length of the array, the highest value in the array, and the lowest value in the array. The function should print out a graph of the array by printing out a * for each value in the appropriate place (below is an example of my printout, which is probably easier than explaining it):

Random Array generated:
6, -2, -4, 5, -3, -4, -3, -1, 5, 2, -2, 0, -7, 2, -3, -4, -3, -1, -5, -3, 1, 7, 3, -7, -7, 3, -8, 1, -5, -4, -2, -5, -8, 0, -4,

Graph of the above array (printed using the function in 6):
8:
7:                     *
6:                     *  *
5:                     *  *
4:                     *  *
3:                     *  *
2:                     *  *
1:                     *  *
0:                     *  *
-1:                     *  *
-2:                     *  *
-3:                     *  *
-4:                     *  *
-5:                     *  *
-6:                     *  *
-7:                     *  *
-8:                     *  *

Filtered Array (using the hanning window):
0, 0, 0, 0, -1, -2, -1, 0, 1, 1, 0, -1, -2, -2, -2, -3, -3, -2, -2, -1, 1, 2, 0, -2, -3, -3, -3, -2, -3, -3, -4, -4, -4, 0, 0,

Graph of the filtered array (printed using function 6):
8:
7:
6:
5:
4:
3:
2:
1:
0:**** * *  *** ****
-1: * *  *  *
-2: *  *****  *  *
14. (10 pts) Write a function that takes as input parameters two integer addresses (both call by pointer). It returns the address of a 2-dimensional array.

In the function, generate a random number between 5 and 10. That will be the size of the array of addresses (call it x). Then generate a second random number between 4 and 8. That will be the size of each array of integers (call it y).

Adjust the input parameters to hold x and y.

Create the 2-dimensional array (of x, y in size), making sure the 2-d array is on the heap.

Fill this array with 0’s. Then generate 5 random number pairs (between 0 and x, and between 0 and y) and place a 1 in each of those random number locations.

Make sure that there isn’t already a 1 in that location (and if there is generate a new x and y random number set).

Return the 2-d array.

In the main, use a for loop (looping x times) to call the print function (function 12) to print out the matrix.