Using multiple variables together in groups

CS 10A – ARRAYS

Introduction to Arrays

- In many program applications, we need a way to collectivize variables quickly when handling bigger volumes of data.
- Arrays are just that: a collection of variables of the same type, differentiated only by a positive integer index value.
- You can create arrays of any variable type, and arrays can be stacked into multiple dimensions.
- The first value of an array starts at 0 and counts up.
  - Arrays have a fixed size, with an index range of [0, size - 1]

Declaring Arrays in C/C++

- Declaring an array is just like declaring a variable, but with braces [ ] included. Braces are reserved symbols for arrays in C/C++.
- There are multiple ways to declare an array, but here are the three methods you'll want to remember the most. (These names have no meaning outside this class!)
  - Method 1: Declare an array of a fixed size
  - Method 2: Manually define each value of the initial array
  - Method 3: Start with a blank slate and specify the size later
    - This last method is done differently between C and C++

Declaring Arrays in C++ - Examples

```
/* Method 1: Inside the braces, use a number (int variables also allowed). This sets the size of the array by number of elements, and the allowed indices are [0, size - 1] */

int arr0[5];            // Array arr0 has 5 elements, an index range of [0, 4]

/* Method 2: No size specified. Instead, specify the initial values of each element to be inside the array, in order, separated by commas inside brackets. The compiler will detect the size from that. Alternatively, you can specify an array size here if it's greater than or equal to the number of initialized elements. */

int arr1[] = {1, 1, 2, 3, 5};      // Array arr1 has 5 elements, an index range of [0, 4]

/* Method 3: Start with a null pointer (don't worry about this concept too much) and any time later declare the actual array size. Useful for when dealing with an unknown number of inputs. */

int * arr2 = NULL; // The asterisk signifies that arr2 is, for now, just a pointer.
arr2 = new int[10]; // The keyword "new" says that arr2 will now be defined as an int array of size 10.
```

Side Note on Method 3

```
int main()                  // Do NOT do the following on any of your assignments
{
        int arr_size;                   // Non-constant integer variable
        cout << “Enter array size: ”;
        cin >> arr_size;                // Get array size from user
        int arr[arr_size];              // Set up new array with user’s value

        // The above method works, but only on SOME compilers
        // Many compilers don’t accept non-constants for Method 1 declarations
        // Use Method 3 as shown on the previous slide instead of the above
        // Method 3 allows non-constant variables to be used for array setups
}
```

Accessing Array Values

Program

```
// State the index you want to use
// Element can be used like other variables
// If referring to the whole array, just use the name
int arr0[] = {5, 3, 9, 0, 8};
int main()
{
        cout << arr0[2] << endl;
        cout << arr0[1] + arr0[4] << endl;
        arr0[3] = 4;
        cout << arr0[3] << endl;
        return 0;
}
```

Console

- ./a.exe

```
9
11
4
```

Determining Array Size

- One of the most important aspects of properly utilizing and processing arrays is knowing its size.
- If you declare this number manually in your code, use a constant int variable to serve as the size value.
- If not, you can also mathematically determine the size inside your code without having to count the values yourself.
  - This is useful when you declare an array with Method 2.
  - Using the sizeof() command, you can have the program automatically calculate the size of a given array.
    - sizeof(input) gives the number of bytes in the given input.
- Using plain numbers in your code to control anything in your program in known as hardcoding, a bad habit to have if you want to spend less time debugging your code.

Determining Array Size - Example

```
const int arr0_size = 7;
double arr0[arr0_size];
// You can now use arr0_size to handle any processes in arr0.

int arr1[] = {3, 4, 6, 9, 11, 14, 0, 12}; // arr1 is expandable
const int arr1_size = sizeof(arr1)/sizeof(arr1[0]);    // arr1_size == 8

/* Gets the number of bytes the array uses, then divides by the
number of bytes the variable itself uses. This gives us the number of
elements inside the array. Doesn't work with arrays declared via
Method 3 for some reason. */
```

Controlling Arrays with Loops

- Since arrays can potentially deal with a massive number of elements, you'll almost always need to use loops to control any and all processes involving arrays.
- Before you do that, you'll need to know the array size, either known from declaration or later calculated.
- Recommended: Use a for loop to handle array processes.
  - for(int i = 0; i < array_size; i++)
    - This will guarantee to cover every element within an array for whatever you would want to use them for.
  - Note that using nested loops to handle arrays will be fairly common.

Forbidden – Writing Outside Array Bounds

- One thing you can do in C++, but should NEVER do, is writing outside the bounds of your array.
  - i.e. In an array of size 5, you can still write to index 5 and above, but really shouldn't.
- Because of the way most computers handle memory, writing outside the predefined bounds of your array has a high risk of accidentally overwriting values in other variables and retrieving incorrect data.
- At best, you'd be introducing all sorts of bugs into your code and at worst, crashing your program.

Safeguarding Arrays

- By using constants or the sizeof() formula to control your array sizes, you can easily avoid bugs that would cause you to read or write outside array bounds.
- Use logic conditions to check that the index you're writing to is within legal limits. Always use index < size_variable to maintain consistency across your array operations.
- In the event you need to use math operations on your index (i.e. arr[i+1]), you can use modulus division with the size to get your index to wrap back to zero.

En Masse Inputs

- Since individual elements inside an array can be treated like separate variables, you can use cin >> array[0] to bring in values one by one, but that alone would be tedious.
- To make the process more efficient, we'll also be using a loop. To quickly get our inputs inside an array, we'll enter all of our input in one line, separated by spaces before hitting the Enter key.
- cin.get() (previously used for pausing) grabs one character at a time from the console. Use this to pick out spaces and new lines from the console.
- To avoid bugs, clear the input stream to get rid of any unused inputs to ensure that cin.get() can pause properly later down the line. You can either use a variant of cin.ignore() on this or getline(). (I prefer the latter.)
- On the next slide, you'll find the code you can copy and paste to handle multiple array inputs. Make sure you understand how it works and why!
- When using Method 3, unused values in array may not start at 0.

Bringing in Input Into Arrays Quickly

Program

```
const int size = 5;
int arr[size], i = 0;
char ch;          // Placeholder variable for cin.get()
string temp;      // Placeholder variable for clearing cin
int main()
{
                  cout << “Enter numbers: ”;
                  do {
                                    cin >> arr[i];
                                    i++;
                                    ch = cin.get();
                  } while(ch != ‘\n’ && i < size);

                  if(i == size && ch != ‘\n’)    // Clear the input stream of excess inputs
                                    getline(cin, temp);
                  else if(i < size && ch == ‘\n’)    // Post-populate unused indices
                                    for(i; i < size; i++)    // Change data as necessary
                                                      arr[i] = 0;
                  for(int k = 0; k < size; k++)
                                    cout << arr[k] << ‘ ‘;
                  return 0;
}
```

Console

- ./a.exe

Enter numbers: 4 1 5 3 2

4 1 5 3 2

- ./a.exe

Enter numbers: 7 1

7 1 0 0 0

- ./a.exe

Enter numbers: 5 3 6 15 9 2 0

5 3 6 15 9

Basic Array Applications

- Since arrays are all about data management, one of the biggest applications you'll learn for arrays is how to sort their contents.
- When you're not busy sorting items, you can quickly perform statistical analysis on arrays.
- Another common application is just simply for saving data. This can be useful for video games with maps.
- Arrays can also be used to manipulate strings for any number of creative applications, such as word games.

Arrays in the Context of Strings

- Strings are conveniently treated as their own separate type of variable in C++. In reality, as it is in C, strings are nothing more than character arrays.
- As such, strings can be accessed and modified the same way as other arrays.
- Each index value inside a string is a character, so it needs to be treated as such.
- Numerous functions in the string library, such as substring, simply automate the use of string indices.

Strings as Arrays - Example

Program

```
string str0 = “I never freeze.”;
int count = 0;
int main()
{
        // Count the number of e’s
        for(int i = 0; i < str0.length(); i++)
                if(str0[i] == ‘e’)
                        count++;
        cout << count << endl;
        return 0;
}
```

Console

➢ ./a.exe

5