CSCI 1300

More Arrays, Vectors and Functions
June 23rd, 2021

Outline

- Strings
  - check characteristics of a string
- Arrays
  - Initialize arrays
  - Indexing arrays
  - Multi dimensional arrays
  - Loops on arrays
- Pseudorandom numbers
- Dry Run examples
- Vectors

Please use the github link for the programing examples and slides.
https://github.com/rahul-aedula95/CSCI-1300

functions to test characteristics of a string

isdigit - returns nonzero if string is a digit (number) (equivalent to true in if conditions)

isalpha - returns nonzero if string has alphabet (equivalent to true in if conditions)

islower - returns nonzero if string is lower case (equivalent to true in if conditions)

isupper - returns nonzero if string is upper case (equivalent to true in if conditions)

Pseudorandom number generation

True random does not exist and often requires a way of generating a random number

Also include the cstdlib library

```
#include<ctime>
int main()
{
int randomNumber;
srand(time(NULL)); // This seeds the rand function.
randomNumber = rand() % 3; // This generates random numbers from 0 to 2. (3 is not included)

return 0;
}
```

Multi dimensional arrays

If you want to create one we can follow:

int arr[3][3] = {{10,20,30},{40,50,60}, {70,80,90}};

This will create the values as shown in the previous example

we can use arr[row_number][column_number] to access an element at some row or column.

Multi dimensional arrays

- We will mostly be dealing with 2 dimensional arrays so lets talk about that since it is easier to visualize.
- Row index and column index can be used to address values

Index	0	1	2
0	10	20	30
1	40	50	60
2	70	80	90

Multi dimensional arrays

- We will mostly be dealing with 2 dimensional arrays so lets talk about that since it is easier to visualize.
- Row index and column index can be used to address values

Index	0	1	2
0	10 arr[0][0]	20 arr[0][1]	30 arr[0][2]
1	40 arr[1][0]	50 arr[1][1]	60 arr[1][2]
2	70 arr[2][0]	80 arr[2][1]	90 arr[2][2]

Arrays

- Similar to strings arrays are basically sequence of other data types
- There are basically consecutively allocated memory for the same data type.
- Visualization of array

Value	10	20	30	40	50	60
Index	0	1	2	3	4	5

Better to store data as arrays rather than use multiple variables to store values.

Loops on arrays

- Since arrays can be very long using loops are very useful to index.
- let us say if an array is int arr[5] = {10,20,30,40,50};

- Since we know the size we can loop over it to access each element individually.
- for (int i = 0; i<array_length;i++) // here our array length is 5

```
{
  cout<<arr[i]<<endl;
}
```

- We can also use this technique on 2d arrays as well.

Initialize an array

If you want to preset values on array

```
int arr[5] = {10,20,30,40,50};
```

You can also create an empty array with some size and allocate values in later.

```
int arr[5];
```

Dry run - example with 1d array

- use example1.cpp from week-4 in our github page
-

i	arr[i]
i → 0	arr[0] → 10

Dry run - example with 1d array

- use example1.cpp from week-4 in our github page
-

i	arr[i]
i → 0	arr[0] → 10
i → 1	arr[1] → 20

Dry run - example with 1d array

- use example1.cpp from week-4 in our github page
-

i	arr[i]
i → 0	arr[0] → 10
i → 1	arr[1] → 20
i → 2	arr[2] → 30

Dry run - example with 1d array

- use example1.cpp from week-4 in our github page
-

i	arr[i]
i → 0	arr[0] → 10
i → 1	arr[1] → 20
i → 2	arr[2] → 30
i → 3	arr[3] → 40

Dry run - example with 1d array

- use example1.cpp from week-4 in our github page
-

i	arr[i]
i → 0	arr[0] → 10
i → 1	arr[1] → 20
i → 2	arr[2] → 30
i → 3	arr[3] → 40
i → 4	arr[4] → 50

Dry run - example with 2d array

- use example2.cpp from week-4 in our github page

i	j	arr[i][j]
i → 0	j → 0	arr[0][0] → 10

Dry run - example with 2d array

- use example2.cpp from week-4 in our github page

i	j	arr[i][j]
i → 0	j → 0	arr[0][0] → 10
i → 0	j → 1	arr[0][1] → 20

Dry run - example with 2d array

- use example2.cpp from week-4 in our github page

i	j	arr[i][j]
i → 0	j → 0	arr[0][0] → 10
i → 0	j → 1	arr[0][1] → 20
i → 0	j → 2	arr[0][2] → 30

Dry run - example with 2d array

- use example2.cpp from week-4 in our github page

i	j	arr[i][j]
i → 0	j → 0	arr[0][0] → 10
i → 0	j → 1	arr[0][1] → 20
i → 0	j → 2	arr[0][2] → 30
i → 1	j → 0	arr[1][0] → 40

Dry run - example with 2d array

- use example2.cpp from week-4 in our github page

i	j	arr[i][j]
i → 0	j → 0	arr[0][0] → 10
i → 0	j → 1	arr[0][1] → 20
i → 0	j → 2	arr[0][2] → 30
i → 1	j → 0	arr[1][0] → 40
i → 1	j → 1	arr[1][1] → 50

Dry run - example with 2d array

- use example2.cpp from week-4 in our github page

i	j	arr[i][j]
i → 0	j → 0	arr[0][0] → 10
i → 0	j → 1	arr[0][1] → 20
i → 0	j → 2	arr[0][2] → 30
i → 1	j → 0	arr[1][0] → 40
i → 1	j → 1	arr[1][1] → 50
i → 1	j → 2	arr[1][2] → 60

Dry run - example with 2d array

- use example2.cpp from week-4 in our github page

i	j	arr[i][j]
i → 0	j → 0	arr[0][0] → 10
i → 0	j → 1	arr[0][1] → 20
i → 0	j → 2	arr[0][2] → 30
i → 1	j → 0	arr[1][0] → 40
i → 1	j → 1	arr[1][1] → 50
i → 1	j → 2	arr[1][2] → 60
i → 2	j → 0	arr[2][0] → 70

Dry run - example with 2d array

- use example2.cpp from week-4 in our github page

i	j	arr[i][j]
i → 0	j → 0	arr[0][0] → 10
i → 0	j → 1	arr[0][1] → 20
i → 0	j → 2	arr[0][2] → 30
i → 1	j → 0	arr[1][0] → 40
i → 1	j → 1	arr[1][1] → 50
i → 1	j → 2	arr[1][2] → 60
i → 2	j → 0	arr[2][0] → 70
i → 2	j → 1	arr[2][1] → 80

Dry run - example with 2d array

- use example2.cpp from week-4 in our github page

i	j	arr[i][j]
i → 0	j → 0	arr[0][0] → 10
i → 0	j → 1	arr[0][1] → 20
i → 0	j → 2	arr[0][2] → 30
i → 1	j → 0	arr[1][0] → 40
i → 1	j → 1	arr[1][1] → 50
i → 1	j → 2	arr[1][2] → 60
i → 2	j → 0	arr[2][0] → 70
i → 2	j → 1	arr[2][1] → 80
i → 2	j → 2	arr[2][2] → 90

Vectors

- Sequence of values similar to arrays but have some useful properties.
  - Advantages:
    - They are of dynamic size, you can add and delete elements on demand unlike arrays which is fixed size.
    - Deallocation of array memory has to be done explicitly unlike vectors which are done automatically.
    - Arrays cannot be returned only referenced from a function, but vectors can be returned as values from a function.

The only thing different is how we initialize and insert values; rest works similar to arrays.

Vectors

- Requires #include<vector>
- syntax:
  - vector<type> variableName(sizeIfRequired)
  - example:
  - vector<int> v(10); // the data type has to be between < and >
- You can create an empty vector:
  - vector<int> v;
- Or fill it with values
  - vector<int> v = {10,20,30};

Some useful built in functions for vectors.

- use example3.cpp to example4.cpp in week-4 in our github page
- You may access elements exactly same as arrays. v[index]
- push_back: element is added to the end of a vector
- pop_back: element is removed from the end of a vector
- clear: clears the vector
- size: returns the size of the vector

Functions

- The best tool to reuse code
- If you need to to use snippets of code instead of rewriting it again you use functions.
- A more mathematical analogy would be to a mathematical function
    - $f(x) \rightarrow y$
    - Here f(x) can be any task which spits out or returns the value y.
    - x is the input parameter or argument
    - y is the output value or returned value
- C++ functions can only return one value.

Three stages of function

- Function declaration: letting the compiler know that there is a function.
- Function definition: Defining the parameters and the task what it needs to do in its own block.
- Calling a function: Using the defined function by giving it some parameters

Exceptions for function scope

- Use testFunc1.cpp to testFunc7.cpp in our github page
- Only if you are passing arrays to a function will the scope and return value will be different as demonstrated in the code snippet in testFunc7.cpp
- This is because when c+= is returning values from an array it actually passes the memory address of variable rather than the value
- This causes any changes to be on the actual variable rather than on just the value.