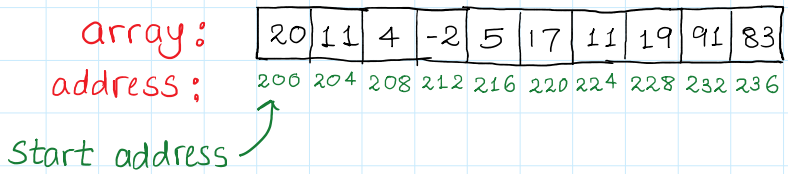
Arrays

18 February 2022 13:12

Arrays help you store variables of the same data type at contiguous memory locations.

Store 10 integers. (Suppose int takes 4 bytes)

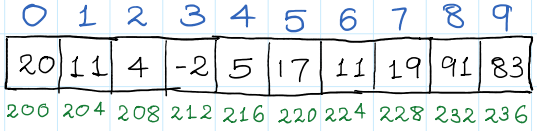


The contiguous memory blocks help us when we try accessing the elements of the array.

Given the start address (address of the array in memory), I can access all the elements by simply incrementing the address by 4 (size of data type).

Suppose our array is named arr. We represent an array as arr[].

To access the elements of an array, we use indices (plural of index). We follow a 0-based indexing in most programming languages



n = 10

∴ arr[0] = 20, arr[6] = 11

Thus, arrays are a 'data structure' that help storing multiple variables of the same data type in contiguous memory locations

DECLARATION OF ARRAYS:

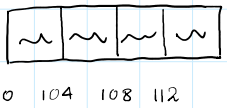
1. Declaring an array to store 100 integers:

```
int arr[100];
```

2. Declaring an array to store 100 booleans:

```
bool isGood[100];
```

Now, in examples ① and ②, arr and isGood are the addresses of the respective arrays and point to the memory location where the first element is stored.

Eg: `int arr[4];` → 

~ : garbage value

where arr[0] is 100 – 103 (4 bytes)
arr[1] is 104 – 107 (4 bytes)
arr[2] is 108 – 111 (4 bytes)
arr[3] is 112 – 115 (4 bytes)

If you print arr, you will get 100 in output.

```
cout << arr;
```
STDOUT: 100

Which is the address of the first element arr[0].

```
cout << &arr[0];
```
STDOUT: 100.

```
cout << arr[1];      // Suppose arr[]:
```

0	1	2	3
4	8	11	1

STDOUT: 8

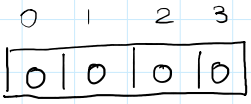
INITIALIZATION:

① `int num[5] = {5,18,40,12}`

0	1	2	3
5	18	40	12

② Initializing an array with 0.

```
int arr[4] = {0};
```



③ Initializing an array with any number. (Homework)

```
int arr[10];
std::fill(arr, arr+10, 27);   // std:: is optional if you have declared the namespace as discussed in previous lectures.
```

start address — end address — value

OR

```
int arr[10];
std::fill_n(arr, 10, 27);
```

start address — No. of elements from the start address to fill — Value

ACCESSING ELEMENTS OUT OF RANGE:

```
//declare
int number[15];

//accessing an array
cout << "Value at 14 index " << number[14] << endl;

cout << "Value at 20 index " << number[20] << endl;
```

⚠ Accessing index out of range will throw an error.

Here, you can only access elements from number[0] to number[14].

```
lovebabbar@192 ~ % cd "/Users/lovebabbar/" && g++ arraysIntro.cpp -o
 arraysIntro && "/Users/lovebabbar/"arraysIntro
arraysIntro.cpp:17:37: warning: array index 20 is past the end of t
e array (which contains 15 elements) [-Warray-bounds]
    cout << "Value at 20 index " << number[20] << endl;
                                    ^      ~~
arraysIntro.cpp:12:5: note: array 'number' declared here
    int number[15];
    ^
1 warning generated.
Value at 14 index 1
Value at 20 index 1
```

PRINTING AN ARRAY:

```
//print the array
for(int i =0; i<n; i++) {
    cout << third[i] << " ";
}
```

where n is the size of the array. See here that our loop is running from 0 to n-1 (not n). Name of our array is third.

Printing array using a function:

```
void printArray(int arr[], int size) {

    cout<< " printing the array " << endl;
    //print the array
    for(int i =0; i<size;i++) {
        cout << arr[i] << " ";
    }
    cout<< " printing DONE " << endl;
}
```

We have to pass the size because int arr[] in the function declaration is simply the address of the array and gives us no information about the size.
Also, if the array has space for 1000 elements but I am storing just 6, then I can pass size as 6 to the function.

MAXIMUM / MINIMUM IN AN ARRAY:

Any value we enter in our array would be less than or equal to max.

```
int getMax(int num[], int n) {

    int max = INT_MIN;

    for(int i = 0; i<n; i++) {
        if(num[i] > max){
            max = num[i];
        }
    }

    //returning max value
    return max;
}
```

```
int getMin(int num[], int n) {

    int min = INT_MAX;

    for(int i = 0; i<n; i++) {
        if(num[i] < min){
            min = num[i];
        }
    }

    //returning min value
    return min;
}
```

```
int main() {

    int size;
    cin >> size;

    int num[100];

    //taking input in array
    for(int i = 0; i<size; i++) {
        cin >> num[i];
    }

    cout << " Maximum value is " << getMax(num, size) << endl;
    cout << " Minimum value is " << getMin(num, size) << endl;


    return 0;
}
```

CAUTION: Never declare a variable-sized array. Always specify a constant size while declaring your array even if it turns out to be very large as compared to your required array size.

ARRAYS ARE ALWAYS PASSED BY REFERENCE:

When we pass an array to a function like this:

func (int arr[]) → address of arr, say 100.

We are not passing a copy of the array unlike variables, since array name 'arr' is the address of the array, thus we pass the address of the array to the function allowing it to make changes to the original array.

This is called passing by reference of array.

HOMEWORK: Print sum of all elements of an array.

```cpp
#include <iostream>
using namespace std;

int getSumOfArray(int arr[], int n) {
    int sum = 0;
    for(int i=0;i<n;i++) {
        sum += arr[i];
    }
    return sum;
}

int main(void)
{
    int n;
    cin >> n;
    int arr[1000];
    for(int i=0;i<n;i++) {
        cin >> arr[i];
    }
    int sum = getSumOfArray(arr, n);
    cout << "Sum of all elements of the array = " << sum << endl;
    return 0;
}
```

```
1  5
2  -4 9 0 11 2
```

Output.txt

```
1  Sum of all elements of the array = 18
2
```

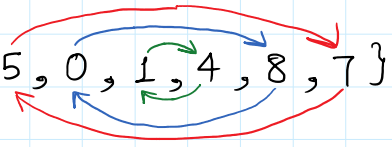
LINEAR SEARCH :

```cpp
bool search(int arr[], int size, int key) {

    for( int i = 0; i<size; i++ ) {

        if( arr[i] == key) {
            return 1;
        }

    }
    return 0;
}
```

Traversing the array and comparing each element with the search key.

REVERSE AN ARRAY :

arr : {5, 0, 1, 4, 8, 7}



{7, 8, 4, 1, 0, 5}

```cpp
void reverse(int arr[], int n) {

    int start = 0;
    int end = n-1;

    while(start<=end) {
        swap(arr[start], arr[end]);
        start++;
        end--;
    }
}
```