Challenge 1 - Insert 1 element in array

1A . inset into empty array

```cpp
#include <iostream>
#include <chrono>
#include "W0-review-arr.hpp"

using namespace std;
using clk =chrono::high_resolution_clock;
volatile int sink_ink = 0;

int main(){
    const int MAX_CAP = 100000;
    int n=0;
    //iniit outsite time block
    int a[10] = {};
    int m = 0;

    auto t0 = clk::now();
    //work here
    input(a,0,1,m,MAX_CAP);

    auto t1 = clk::now();
    cout<< chrono::duration_cast<chrono::nanoseconds>(t1-
}
```

```
void insert(int* ptr_arr, int pos, int val, int &n, int size =
    if (not ptr_arr){
        cout<<"array is empy"<<endl;
        return;
    }
    if (n >= size){
        cout << "Error! Array is full." << endl;
        return;
    }
    if (pos < 0 || pos > n){
        cout << "Invalid position." << endl;
        return;
    }
    for (int i = n; i > pos; i--) {
        ptr_arr[i] = ptr_arr[i - 1];
    }
    ptr_arr[pos] = val;
    n++;
}
```

Run	Time
1	142
2	115
3	167
4	132
5	123

Time measure in Nanoseconds
average : 135.8

explanation :
inserting into an empty array require no shifting of elements. only one operation is required so that's why it's fast.

1B . insert when not full, no index specified

```
 9   int main(){
10       const int MAX_CAP = 100000;
11       //iniit outsite time block
12       int a[MAX_CAP] = {0};
13       int n = MAX_CAP/2;
14
15       auto t0 = clk::now();
16       //work here
17       insertAtBeginning(a,9,n,MAX_CAP);
18
19       auto t1 = clk::now();
20       cout<< chrono::duration_cast<chrono::microseconds>(t
21   }
```

Run	Time
1	143
2	168
3	121
4	117
5	116

Time measures in Microsecond
Average = 133

explanation:
Inserting at the beginning of an array require the program to shif all other element to the right.That is why this operation take a lot of time. If the array is empty like in 1A the program do not need to shift any element and that's why it is much faster.

```
59  void insertatlast(int* ptr_arr,int val , int &n, int size
60      if (n>size){
61          cout<<"the array is full"<<endl;
62      }
63      ptr_arr[n]=val;
64  }
```

```
 9  int main(){
10      const int MAX_CAP = 100000;
11      //iniit outsite time block
12      int a[MAX_CAP] = {0};
13      int n = MAX_CAP/2;
14
15      auto t0 = clk::now();
16      //work here
17      insertatlast(a,10,n,MAX_CAP);
18
19      auto t1 = clk::now();
20      cout<< chrono::duration_cast<chrono::microseconds>(t
21  }
```

Run	Time
1	186
2	137
3	215
4	142
5	148

Time measures in Nanosecond
Average = 165.5

explanation:
Inserting at the end of an array do not require any shifting of elements and there's only one operation to

do. This function take constant amout of time to run regarding the size of the array.

1C. Insert at given index

```
int main(){
    const int MAX_CAP = 100000;
    //iniit outsite time block
    int a[MAX_CAP] = {0};
    int n = MAX_CAP/2;

    auto t0 = clk::now();
    //work here
    insert(a,0,10,n,MAX_CAP);

    auto t1 = clk::now();
    cout<< chrono::duration_cast<chrono::microseconds>(t
}
```

```
int main(){
    const int MAX_CAP = 100000;
    //iniit outsite time block
    int a[MAX_CAP] = {0};
    int n = MAX_CAP/2;

    auto t0 = clk::now();
    //work here
    insert(a,n,10,n,MAX_CAP);

    auto t1 = clk::now();
    cout<< chrono::duration_cast<chrono::microseconds>(t
}
```

```cpp
int main(){
    const int MAX_CAP = 100000;
    //iniit outsite time block
    int a[MAX_CAP] = {0};
    int n = MAX_CAP/2;

    auto t0 = clk::now();
    //work here
    insert(a,n/2,10,n,MAX_CAP);

    auto t1 = clk::now();
    cout<< chrono::duration_cast<chrono::microseconds>(t
}
```

Run	beginning	middle	last
1	84	71	0
2	84	48	0
3	96	49	0
4	117	48	0
5	116	78	0
average	99.4	58.8	0

Time measures in Microseconds

explanation:
Insert at the beginning is the slowest because it require shifting the most element. while inserting in the middle is a bit faster becuase
it require the program to shift less amout of elements and inserting at the last index is the fastest because it require no element shifting.

1D. insert when full(resize)

D1: We can’t insert without overwrite becuase the array is full and all the slot are occupied.

D2 : simulate dynamic array

```c
int* insertfullarray(int* ptr_arr,int val,int size){
    int new_size = size *2;
    // allocated new array
    int* new_arr = (int*)malloc(new_size * sizeof(int));
    //copy the data to new array
    int i =0;
    for (i =0;i<size;i++){
        new_arr[i] = ptr_arr[i];
    }
    // insert the value at last inddex
    new_arr[i] = val;
    return new_arr;
}
```

```c
int* insertfullarray(int* ptr_arr,int val,int pos, int s
    int new_size = size *2;
    // allocated new array
    int* new_arr = (int*)malloc(new_size * sizeof(int));
    //copy the data to new array
    int i =0;
    for (i =0;i<size;i++){
        new_arr[i] = ptr_arr[i];
    }
    // insert the value
    for (int i = size; i > pos; i--) {
        ptr_arr[i] = ptr_arr[i - 1];
    }
    ptr_arr[pos] = val;
    return new_arr;
}
```

1B

Run	Time
1	143
2	168
3	121
4	117
5	116

1C

Run	beginning	middle	last
1	84	71	0
2	84	48	0
3	96	49	0
4	117	48	0
5	116	78	0
average	99.4	58.8	0

1D

Run	Time
1	377
2	384
3	392
4	405
5	364

time measure in microseconds
average = 384.4

explanation:

compare to 1B and 1C this take more time because the extra time come from:
- calculate new size
- allocate new array
- copy all the value to new array
- insert the value into the array

Challenge 2:
2A Remove from the middle

```
 9   int main(){
10       const int MAX_CAP = 100000;
11       //iniit outsite time block
12       int a[MAX_CAP] = {1,2,3,4,5};
13       int n = MAX_CAP/2;
14
15       auto t0 = clk::now();
16       //work here
17       deletes(a,n/2,n);
18
19       auto t1 = clk::now();
20       cout<< chrono::duration_cast<chrono::microseconds>(t
21   }
```

Run	Time
1	86
2	64
3	62
4	69
5	64

time measure in microseconds
average : 69

explanation :
The time measured is the time it take to shift n/2 elements to the left.

2B. remove the last index

```
 9   int main(){
10       const int MAX_CAP = 100000;
11       //iniit outsite time block
12       int a[MAX_CAP] = {1,2,3,4,5};
13       int n = MAX_CAP/2;
14
15       auto t0 = clk::now();
16       //work here
17       deletes(a,n,n);
18
19       auto t1 = clk::now();
20       cout<< chrono::duration_cast<chrono::nanoseconds>(t1-
21   }
```

Run	Time
1	200
2	165
3	177
4	212
5	162

time measure in nanosecond
average : 183.2

explanation:
It's really faster than the previous one bacause there is no shifting elements. The only thing the program need to do is set the value at the positions to 0.

2C. remove from the beginninng

```
9   int main(){
10      const int MAX_CAP = 100000;
11      //iniit outsite time block
12      int a[MAX_CAP] = {1,2,3,4,5};
13      int n = MAX_CAP/2;
14
15      auto t0 = clk::now();
16      //work here
17      deletes(a,0,n,MAX_CANON);
18
19      auto t1 = clk::now();
20      cout<< chrono::duration_cast<chrono::nanoseconds>(t1-
21  }
```

Run	Time
1	224
2	154
3	143
4	143
5	135

Time measure in Microseconds
Average : 159.8

explanation:
This one is much slower than the previous two because this program need to shift all n/2 elements in the array so the left.

2D. edge sanity

```
int main(){
    const int MAX_CAP = 100000;
    //iniit outsite time block
    int a[1] = {1};
    int n = 1;

    auto t0 = clk::now();
    //work here
    deletes(a,0,n,1);

    auto t1 = clk::now();
    cout<< chrono::duration_cast<chrono::nanoseconds>(t1-
}
```

Run	Time
1	142
2	148
3	121
4	134
5	130

Time measure in Nanoseconds
Average : 135

Explanation and compare:
This program is really fast becuse it do not need to shift any thing. compare to 2C this program is about one thousand time faster and compare to 2B it's about the same since it both do the same operation.

Challenge 3:
-
When accessing an array arr[i] in C++ do not require

scanning. It is a constant time operation regarding the number of element in the array arr[i] will take the same time to process.
Example : we have arr[100000], accessing arr[100] and arr[10000] will take the same amount of time.

For linear search in the best case the thing we want to search for is in the first index of array so the operation take roughly the same time as accessing array but In the Worst Case the target is in the last index of the array so the time to execute it will grow by n.
Example: we have arr[10000], performing linear search with best case will take O(1) and the worst case will take O(n)