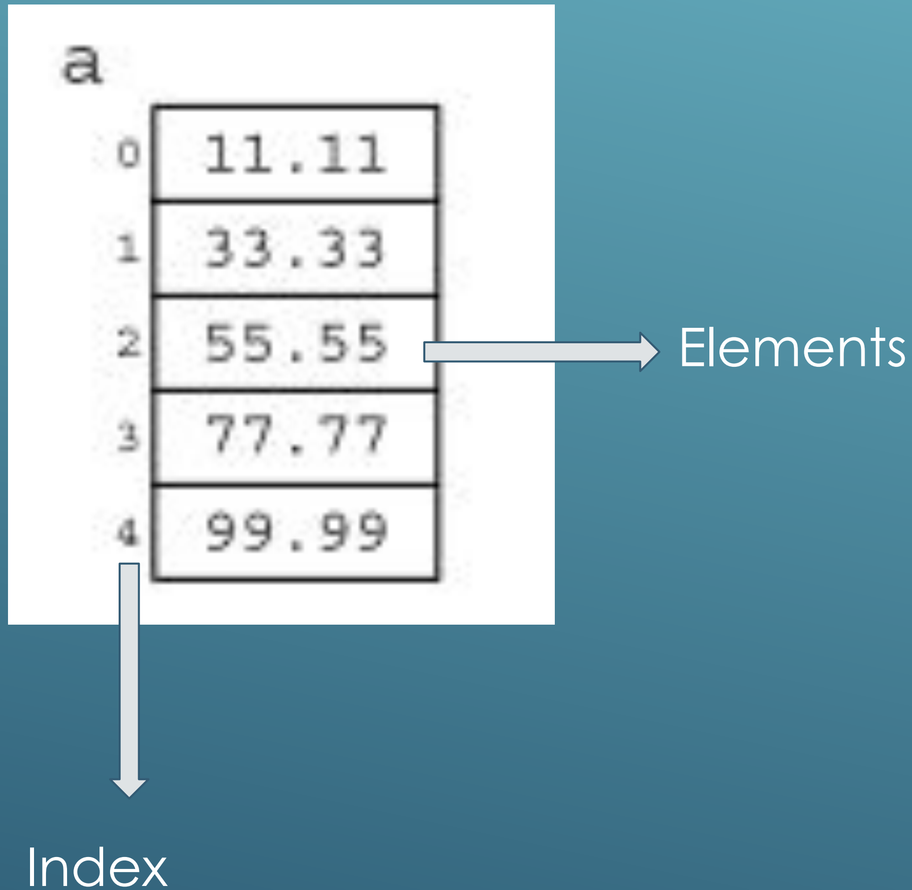
Programming 1 (C++)

Arrays

INTRODUCTION

- An array is a sequence of objects all of which have the same type




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

int main() {
    // Example of declaring an array of integers
    int numbers[5]; // array of 5 integers

    // Accessing elements of the array
    numbers[0] = 10;
    numbers[1] = 20;
    // ...

    return 0;
}
```

- The method of numbering the ith element with index i–1 is called zero-based indexing

PROCESSING ARRAYS

- Processing arrays involves performing operations on each element of the array, such as printing, modifying, or calculating values.

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

int main() {
    int numbers[5] = {10, 20, 30, 40, 50};

    // Printing elements of the array
    for(int i = 0; i < 5; ++i) {
        cout << numbers[i] << " ";
    }

    return 0;
}
```

PROCESSING ARRAYS

- An array is a composite object: it is composed of several elements with independent values. In contrast, an ordinary variable of a primitive type is called a scalar object.

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

int main()
{ double a[3];
a[2] = 55.55;
a[0] = 11.11;
a[1] = 33.33;
cout << "a[0]="<< a[0] << endl;
cout << "a[1]="<< a[1] << endl;
cout << "a[2]="<< a[2] << endl;
}
```

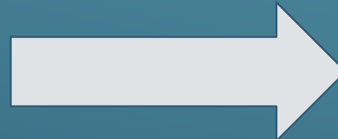
INITIALIZING AN ARRAY

- Arrays can be initialized during declaration using curly braces {} with comma-separated values.

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

- An array can be “zeroed out” by declaring it with an initializer list together with an explicit size value

```
int main()
{ float a[7] = { 22.2,44.4,66.6 };
int size = sizeof(a)/sizeof(float);
for (int i=0; i<size; i++)
cout << "\ta[" << i << "]="<< a[i] << endl;
}
```



a

0	55.5
1	66.6
2	77.7
3	0.0
4	0.0
6	0.0
7	0.0

AN UNINITIALIZED ARRAYS

- An initialization is not the same as an assignment. Arrays can be initialized, but they cannot be assigned

```
float a[7] = { 22.2,44.4,66.6 };
float b[7] = { 33.3,55.5,77.7 };
b=a; // ERROR: arrays cannot be assigned!
```

- Nor can an array be used to initialize another array:

```
float a[7] = { 22.2,44.4,66.6 };
float b[7] = a; // ERROR: arrays cannot be used as initializers!
```

ARRAY INDEX OUT OF BOUNDS

- In some programming languages, an index variable will not be allowed to go beyond the bounds set by the array's definition. For example, in Pascal, if an array a is defined to be indexed from 0 to 3, then the reference a[6] will crash the program.
- This is a security device that does not exist for arrays in C++ (or C). As the next example shows, the index variable may run far beyond its defined range without any error being detected by the computer.

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

int main() {
    int arr[5] = {1, 2, 3, 4, 5};

    // Accessing an element outside the bounds of the array
    // This will cause undefined behavior
    cout << arr[4] << endl; // Out of bounds access

    return 0;
}
```

ARRAY INDEX OUT OF BOUNDS

- It is the C++ programmer's responsibility to ensure that array index values are kept in range

```
int main()
{ const int SIZE=4;
float a[] = { 22.2,44.4, 66.6 };
float x=11.1;
cout << "x="<<x << endl;
a[3333] = 88.8; // ERROR: index is out of bounds!
cout << "x="<<x << endl;
}
```

PASSING ARRAY TO A FUNCTION

- Arrays can be passed to functions either by passing the array name or by passing a pointer to the array.

```
int sum(int[],int);
int main()
{ int a[] = { 11,33, 55,77 };
int size = sizeof(a)/sizeof(int);
cout << "sum(a,size) = " << sum(a,size) << endl;
}
int sum(int a[],int n)
{ int sum=0;
for (int i=0; i<n; i++)
sum += a[i];
return sum;
}
```

PASSING ARRAY TO A FUNCTION

- Arrays can be passed to functions either by passing the array name or by passing a pointer to the array.

```
void printArray(int arr[], int size) {
    for(int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
}
int main() {
    int numbers[5] = {10, 20, 30, 40, 50};
    printArray(numbers, 5);
    return 0;
}
```

LINEAR SEARCH ALGORITHM

- Linear search is a simple searching algorithm that searches for a target value in an array by sequentially checking each element from the beginning.

```
int index(int,int[],int);
int main()
{ int a[] = { 22,44, 66,88,44,66,55 };
cout << "index(44,a,7)="<< index(44,a,7) << endl;
cout << "index(50,a,7)="<< index(50,a,7) << endl;
}
int index(int x,int a[],int n)
{ for (int i=0; i<n; i++)
if (a[i] == x) return i;
return n; // x not found
}
```

BUBBLE SORT ALGORITHM

- The Linear Search algorithm is not very efficient.
- Although not as efficient as most others, the Bubble Sort is one of the simplest sorting algorithms.

```
void print(float[],int);
void sort(float[],int);
int main()
{ float a[] = {55.5, 22.5,99.9,66.6,44.4,88.8,33.3,77.7};
print(a,8);
sort(a,8);
print(a,8);
}
void sort(float a[],int n)
{ // bubble sort:
for (int i=1; i<n; i++)
// bubble up max{a[0..n-i]}:
for (int j=0; j<n-i; j++)
if (a[j] > a[j+1]) swap(a[j],a[j+1]);
// INVARIANT: a[n-1-i..n-1] is sorted
}
```

BINARY SEARCH ALGORITHM

- The binary search uses the “divide and conquer” strategy.
- It repeatedly divides the array into two pieces and then searches the piece that could contain the target value.

```
int index(int,int[],int);
int main()
{ int a[] = { 22,33, 44,55,66,77,88 };
cout << "index(44,a,7)="<< index(44,a,7) << endl;
cout << "index(60,a,7)="<< index(60,a,7) << endl;
}
int index(int x,int a[],int n)
{// PRECONDITION: a[0] <= a[1] <= ... <= a[n-1];
// binary search:
int lo=0,hi=n-1,i;
while (lo <= hi)
{ i = (lo + hi)/2; // the average of lo and hi
if (a[i] == x) return i;
if (a[i] < x) lo = i+1; // continue search in a[i+1..hi]
else hi = i-1;
// continue search in a[lo..i-1]
}
return n; // x was not found in a[0..n-1]
}
```

USING ARRAYS WITH ENUMERATION TYPES

- Enumeration types provide a way to define named constants, which can be used as indices for arrays.
- The advantage of using enumeration constants this way is that they render your code “self-documenting.”

```
int main()
{ enum Day { SUN,MON,TUE,WED,THU,FRI,SAT };
float high[SAT+1] = {88.3,95.0,91.2,89.9,91.4,92.5,86.7};
for (int day = SUN; day <= SAT; day++)
cout << "The high temperature for day " << day
<< " was " << high[day] << endl;
}
```

TYPE DEFINITIONS

- Type definitions can be used to simplify complex array declarations, improving code readability.

```
typedef float Sequence[];
void sort(Sequence,int);
void print(Sequence,int);
int main()
{ Sequence a = {55.5, 22.5,99.9,66.6,44.4,88.8,33.3,77.7};
print(a,8);
sort(a,8);
print(a,8);
}
void sort(Sequence a, int n)
{ for (int i=n-1; i>0; i--)
for (int j=0; j<i; j++)
if (a[j] > a[j+1]) swap(a[j],a[j+1]);
}
```

MULTIDIMENSIONAL ARRAYS

- Multidimensional arrays are arrays of arrays, commonly used to represent tables or matrices.

```
void read(int a[ ][5]);
void print(cont int a[ ][5]);
int main()
{ int a[3][5];
read(a);
print(a);
}
void read(int a[][5])
{ cout << "Enter 15 integers,5 per row:\n";
for (int i=0; i<3; i++)
{ cout << "Row " << i << ": ";
for (int j=0; j<5; j++)
cin >> a[i][j];
}
}
void print(const int a[][5])
{ for (int i=0; i<3; i++)
{ for (int j=0; j<5; j++)
cout <<""<< a[i][j];
cout << endl;
}
}
```

```
Enter 15 integers, 5 per row:
Row 0: 44 77 33 11 44
Row 1: 60 50 30 90 70
Row 2: 85 25 45 45 55
 44 77 33 11 44
 60 50 30 90 70
 85 25 45 45 55
```

THANK YOU