Section 3

Introduction to Arrays

Presentation by *Asem Alaa*

Introduction to Arrays

Introduction to Arrays

Arrays structures, are:

Introduction to Arrays

Arrays structures, are:

- The **simplest** data structure.

Introduction to Arrays

Arrays structures, are:

- The **simplest** data structure.
- Placed **contiguously** in memory.

Introduction to Arrays

Arrays structures, are:

- The **simplest** data structure.
- Placed **contiguously** in memory.
- Referred by the **address of the first element**.

Introduction to Arrays

Arrays structures, are:

- The **simplest** data structure.
- Placed **contiguously** in memory.
- Referred by the **address of the first element**.
- **Simple** iterations using for-loops.

Introduction to Arrays

Arrays structures, are:

- The **simplest** data structure.
- Placed **contiguously** in memory.
- Referred by the **address of the first element**.
- **Simple** iterations using for-loops.

Arrays can be constructed on:

Introduction to Arrays

Arrays structures, are:

- The **simplest** data structure.
- Placed **contiguously** in memory.
- Referred by the **address of the first element**.
- **Simple** iterations using for-loops.

Arrays can be constructed on:

- Stack Memory => **Static Arrays**

Introduction to Arrays

Arrays structures, are:

- The **simplest** data structure.
- Placed **contiguously** in memory.
- Referred by the **address of the first element**.
- **Simple** iterations using for-loops.

Arrays can be constructed on:

- Stack Memory => **Static Arrays**
- Heap Memory => **Dynamic Arrays**

Static Arrays

Static Arrays

- **Limited** in size.

Static Arrays

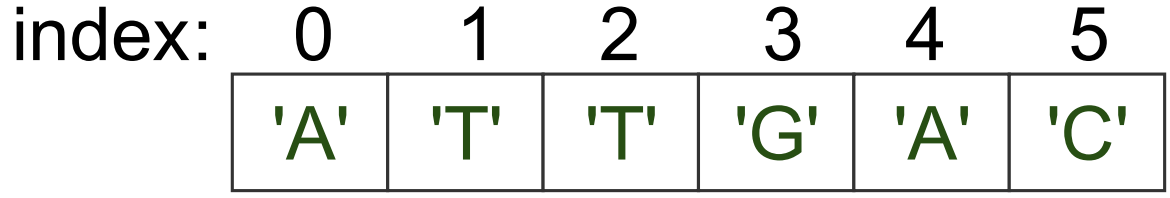
- **Limited** in size.
- **Size** determined at **compile-time**.

Static Arrays

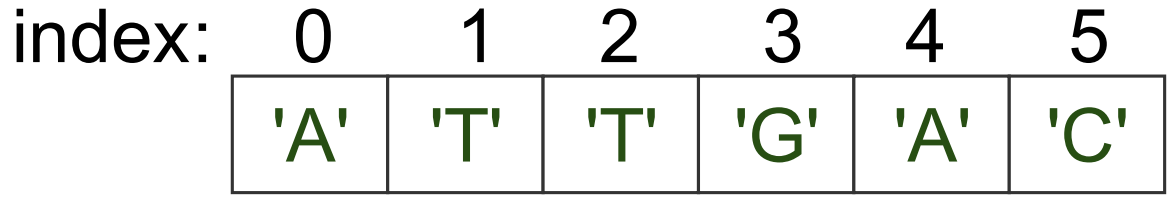
- **Limited** in size.
- **Size** determined at **compile-time**.
- Automatic memory management.

Constructing Static Array

Constructing Static Array



Constructing Static Array




```
// Construction of array-of-integers with size 10.
int array1[10];

// Construction of array-of-characters with size 150.
char array2[150];

// Construction + Initialization of array-of-doubles with size 4
double physicalConstants[] = { 3.1415926 , 2.717 , 1.618 , 1.0 };

// Construction + Initialization of array-of-characters of size 6
char dna[] = { 'A' , 'A' , 'C' , 'T' , 'G' , 'C' };
```

Accessing Elements of Array

Accessing Elements of Array

```
double a[10]; // Declaration
```

Accessing Elements of Array

```
double a[10]; // Declaration
```

To access array elements,

Accessing Elements of Array

```
double a[10]; // Declaration
```

To access array elements,

- First element => `a[0]`.

Accessing Elements of Array

```
double a[10]; // Declaration
```

To access array elements,

- First element => `a[0]`.
- Base pointer => the address of first element => `&a[0]`.

Accessing Elements of Array

```
double a[10]; // Declaration
```

To access array elements,

- First element => `a[0]`.
- Base pointer => the address of first element => `&a[0]`.
- Second element => `a[1]`.

Accessing Elements of Array

```
double a[10]; // Declaration
```

To access array elements,

- First element => `a[0]`.
- Base pointer => the address of first element => `&a[0]`.
- Second element => `a[1]`.
- index = offset = distance from `a[0]`.

Example: Factorials Sequence

Let `factorial` an integer array holding a lookup table for factorial numbers

Example: Factorials Sequence

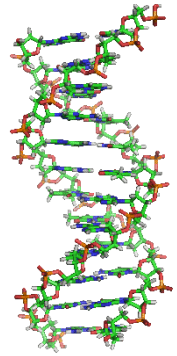
Let `factorial` an integer array holding a lookup table for factorial numbers

```
int factorial[5];

factorial[0] = 1;
factorial[1] = 1;
factorial[2] = 2 * factorial[1];
factorial[3] = 3 * factorial[2];
factorial[4] = 4 * factorial[3];
```

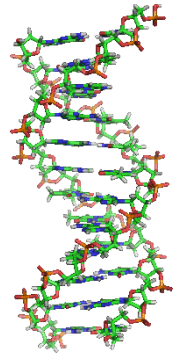
Example: DNA Sequence

class: left, top Let `dna` a sequence of some genetic region.



Example: DNA Sequence

class: left, top Let `dna` a sequence of some genetic region.



```
// Alternative way of Construction + Initialization
// of array-of-characters of size 6
char dna[] = { 'A' , 'A' , 'C' , 'T' , 'G' , 'C' };

std::cout << dna[0] << std::endl; // Prints: A

dna[1] = 'T'; // Modifies the second element to 'T'.

std::cout << dna[1] << std::endl; // Prints: T
```

Iterating Over Static Array

Iterating Over Static Array

```
for( int i = 0; i < 6 ; ++i )
{
    std::cout << dna[i] << ", ";
}

std::cout << "\n";
```

Application: Compute Average of Array

Application: Compute Average of Array

Implement the following mean function (logic), to calculate the average of array elements.

Application: Compute Average of Array

Implement the following mean function (logic), to calculate the average of array elements.

$$\bar{x} = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right) = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

Application: Compute Average of Array

Implement the following mean function (logic), to calculate the average of array elements.

$$\bar{x} = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right) = \frac{x_1 + x_2 + \cdots + x_n}{n}$$

```
double mean( double *array , int size )
{
    double sum = 0;
    for( int i = 0 ; i < size ; ++i )
        sum = sum + array[ i ];
    return sum / size;
}
int main()
{
    double ecg_samples[] = { 9.1 , 12.9, 12.4, 15.2, 19.0, 23.3 };
    double ecg_mean = mean( &ecg_samples[0] , 6 );
}
```

Dynamic Arrays

Dynamic Arrays

- Lives on **Heap Memory**.

Dynamic Arrays

- Lives on **Heap Memory**.
- **Flexibility**: **Size** determined at compilation or run-time .

Dynamic Arrays

- Lives on **Heap Memory**.
- **Flexibility**: **Size** determined at compilation or run-time .
- You can construct **very large**.

Dynamic Arrays

- Lives on **Heap Memory**.
- **Flexibility**: **Size** determined at compilation or run-time .
- You can construct **very large**.
- You **need to manually delete dynamic arrays**.

Constructing Dynamic Array

```
// Construction of array-of-integers with arbitrary.
int size = 0;
std::cin >> size; // size determined at run-time.

// You cannot construct static arrays with an arbitrary size
// like in dynamic array.
int *array1 = new int[ size ];

// Construction of array-of-characters with size 150000
// (around 150 Mega Bytes in memory).
char dna_chromosome11 = new char[ 150000 ];
```

Constructing Dynamic Array

```
// Construction of array-of-integers with arbitrary.
int size = 0;
std::cin >> size; // size determined at run-time.

// You cannot construct static arrays with an arbitrary size
// like in dynamic array.
int *array1 = new int[ size ];

// Construction of array-of-characters with size 150000
// (around 150 Mega Bytes in memory).
char dna_chromosome11 = new char[ 150000 ];
```

Any typo? 

Constructing Dynamic Array

```
// Construction of array-of-integers with arbitrary.
int size = 0;
std::cin >> size; // size determined at run-time.

// You cannot construct static arrays with an arbitrary size
// like in dynamic array.
int *array1 = new int[ size ];

// Construction of array-of-characters with size 150000
// (around 150 Mega Bytes in memory).
char *dna_chromosome11 = new char[ 150000 ];
```

Any typo? 

Memory Management

```
int *array1 = new int[ 900 ];

char *dna_chromosome11 = new char[ 150000 ];

// Load some DNA from external file to the constructed array
loadDNA( dna_chromosome11 , 150000 ,
        "/home/user/chromosomes/some-dna.txt");

// Do some interesting analysis on your genome.
someInterestingFunction( &dna_chromosome11[0] , 150000 );

// Another operations on array1
anotherInterestingFunction( &array1[0] , 900 );

// After we no longer need array1,
delete [] array1; // Note the square brackets!
delete [] dna_chromosome11;
```

Special Case: Array of Characters (String)

```
// Alternative way of Construction + Initialization of
// array-of-characters of size 6
char dna[] = { 'A' , 'A' , 'C' , 'T' , 'G' , 'C' , '\0'};

std::cout << dna << "\n"; // Prints: AACTGC

// Alternatively, it is always recommended to use 'std::string'
std::string dna2 = "AACTGC"
std::cout << dna2 << "\n";
```

Basic Operations on Static and Dynamic Arrays

Copying between arrays

Basic Operations on Static and Dynamic Arrays

Copying between arrays

Assume that you want to copy an array to another array (either static or dynamic).

```
#include <algorithm> // Needed for std::copy
#include <iostream> // Needed for std::cout
int main()
{
    char dna1[] = { 'A' , 'A' , 'C' , 'T' , 'G' , 'C' , '\0'};

    char dna2[ 7 ];

    std::copy( &dna1[0] , &dna1[6] , &dna2[0] );

    std::cout << dna2 << std::endl;
}
```

std::copy

std::copy

```
std::copy( &dna1[0] , &dna1[6] , &dna2[0] );
```

std::copy

```
std::copy( &dna1[0] , &dna1[6] , &dna2[0] );
```

To copy from **source** array to **target** array:

std::copy

```
std::copy( &dna1[0] , &dna1[6] , &dna2[0] );
```

To copy from **source** array to **target** array:

1. Address of first element of **source** array.

std::copy

```
std::copy( &dna1[0] , &dna1[6] , &dna2[0] );
```

To copy from **source** array to **target** array:

1. Address of first element of **source** array.
2. Address of last element of **source** array.

std::copy

```
std::copy( &dna1[0] , &dna1[6] , &dna2[0] );
```

To copy from **source** array to **target** array:

1. Address of first element of **source** array.
2. Address of last element of **source** array.
3. Address of first element of **target** array.

std::copy

```
std::copy( &dna1[0] , &dna1[6] , &dna2[0] );
```

To copy from **source** array to **target** array:

1. Address of first element of **source** array.
2. Address of last element of **source** array.
3. Address of first element of **target** array.

Equivalent to:

```
for( int i = 0 ; i < 7 ; ++i )
{
    dna2[i] = dna1[i];
}
```

Arrays ∩ `struct`

Revisiting `struct`

Arrays ∩ `struct`

Revisiting `struct`

```
double area( double w , double h )
{
    return w * h;
}
```

Arrays ∩ `struct`

Revisiting `struct`

```
double area( double w , double h )
{
    return w * h;
}
```

```
struct Rectangle
{
    double w;
    double h;
};
```

Revisiting struct

```
double area( double w , double h )
{
    return w * h;
}
```

```
struct Rectangle
{
    double w;
    double h;
};
```

- Rectangle is now a user-defined type,

Revisiting struct

```
double area( double w , double h )
{
    return w * h;
}
```

```
struct Rectangle
{
    double w;
    double h;
};
```

- Rectangle is now a user-defined type,
- consists of two doubles.

Revisiting struct (cont'd)

```
struct Rectangle
{
    double w; // First member
    double h; // Second member
}; // Don't forget a semicolon here!

double area( Rectangle rectangle )
{
    return rectangle.w * rectangle.h;
}
// By the way..
double area2( Rectangle *prect )
{
    return prect->w * prect->h;
}
```

Revisiting struct (cont'd)

```
int main()
{
    Rectangle rect;
    rect.w = 3;
    rect.h = 5;

    std::cout << area( rect ) << std::endl;
    std::cout << area2( &rect ) << std::endl;
    return 0;
}
```

Arrays ∩ `struct`

Consider a function that returns the summation of array.

Arrays ∩ struct

Consider a function that returns the summation of array.

```
int sum( int *arr, int size )
{
    int sum = 0;
    for( int i = 0; i < size ; ++i )
    {
        sum += arr[ i ];
    }
    return sum;
}
```

Arrays ∩ struct

Consider a function that returns the summation of array.

```
int sum( int *arr, int size )
{
    int sum = 0;
    for( int i = 0; i < size ; ++i )
    {
        sum += arr[ i ];
    }
    return sum;
}
```

Can we do better?

Arrays ∩ `struct`

We may also package an array with its size, using `struct`

Arrays ∩ `struct`

We may also package an array with its size, using `struct`

```
struct IntegerArray
{
    int *data;
    int size;
};

int sum( IntegerArray array )
{
    int sum = 0;
    for( int i = 0; i < array.size ; ++i )
    {
        sum += array.data[ i ];
    }
    return sum;
}
```

Arrays ∩ struct

```
int main()
{
    IntegerArray array;
    array.data = new int[10];
    array.size = 10;
    std::cout << sum( array ) << std::endl;

    // We still need to delete the array on the heap
    delete [] array.data;
}
```

`struct` for returning multiple values

Example 1: Find the roots

$$ax^2 + bx + c = 0$$

Recall the exercise of lab 2..

```
void root( double a, double b, double c, double &x1, double &x2)
{
    double delta = std::sqrt( b*b - 4*a*c);
    x1 = (-b - delta)/(2*a);
    x2 = (-b + delta)/(2*a);
}
int main(int argc, char **argv)
{
    double a,b,c,x1,x2; std::cin >> a >> b >> c;
    root(a, b, c, x1, x2);
    std::cout << x1 << "\n" << x2 << "\n";
}
```

`struct` for returning multiple values

Example 1: Find the roots

```
struct Roots
{
    double x1; double x2;
};
Roots root( double a, double b, double c)
{
    Roots r;
    double delta = std::sqrt( b*b - 4*a*c);
    r.x1 = (-b - delta)/(2*a);
    r.x2 = (-b + delta)/(2*a);
    return r;
}
int main()
{
    double a,b,c; std::cin >> a >> b >> c;
    Roots r = root(a, b, c);
    std::cout << r.x1 << "\n" << r.x2 << "\n";
}// Try online: http://cpp.sh/9tkdv
```

`struct` for returning multiple values

Example 2: ECG statistics

`struct` for returning multiple values

Example 2: ECG statistics

```
struct ECGArray // We could name it also DoubleArray
{
    double *data;
    int size;
}

struct Statistics
{
    double mean;
    double variance;
    double min;
    double max;
}
```

`struct` for returning multiple values

Example 2: ECG statistics (cont'd)

```
// Very self-explaining function header!
Statistics analyzeECG( ECGArray ecg )
{
    Statistics analysis;

    analysis.mean = // Some logic here
    analysis.variance = // Some logic there
    analysis.max = //
    analysis.min = //
    return analysis;
}
```

Thank you