1 Array Basics 3
2 Iterating Over Arrays 5
3 Arrays as Function Parameters 7
4 Multi-Dimensional Arrays 9
5 Strings 10
• We would like to be able to store a sequence of values all of the same type and then perform operations on this sequence

• We already saw how to implement a sequence of values using a chain of nodes; each node is a struct with a value and a next pointer

• Arrays are an alternative approach where the sequence of values is directly mapped into a linear sequence of variables
1. Array Basics

- Arrays require introducing new types and new operators
- Every type T has a corresponding array type
- T name[size] declares an array of size elements each of type T

```plaintext
1. int a[4]; // array of four ints
2. char b[4]; // array of four chars
3. float c[4]; // array of four floats
```

- size can be const or non-const, but const is much more efficient

```plaintext
1. const int a_size = 4; // size is a const variable
2. int a[a_size]; // array of four ints
3. int b_size = 4; // size is a non-const variable
4. int b[b_size]; // array of four ints
```

- Can initialize an array with struct-like initialization syntax

```plaintext
1. int a[] = { 10, 11, 12, 13 };
```

- Cannot assign to an array, only to array elements

```plaintext
1. int a[4] = { 10, 11, 12, 13 }; // array of four ints
2. int b[4]; // array of four ints
3. b = a; // illegal!
```
1. Array Basics

- The **subscript** operator ([i]) evaluates to the value of element i
- The subscript operator is used for reading/writing elements
- The subscript operator uses zero-based indexing

```c
1  int a[4]; // array of four ints
2  a[0] = 13; // assign 13 to element 0
3  int b = a[0]; // initialize b with value of element 0
```

- Out-of-bounds access is perfectly legal and dangerous!

```c
1  int a[4]; // array of four ints
3  int b = a[5]; // out-of-bounds read!
```

**Example declaring, initializing, RHS and LHS subscripting**

```c
1  int a[4] = { 10, 11, 12, 13 };
2  int b = a[0] + a[1];
3  a[2] = b;
4  a[3] = a[0];
```

**Relationship between arrays and pointers**

- Assume we declare an array `int a[4]`
- Type of the expression `a` is an “array of four ints”
- Expression `a` can **act** like a pointer to first element in the array
- Similarly, a pointer can sometimes **act** like an array
- Definitely one of the more inconsistent aspects of C syntax
2. Iterating Over Arrays

- We primarily work with arrays by iterating over their elements
- Example of calculating average of an array of ints

```c
int a[] = { 10, 20, 30, 40 };  // Stack
int sum = 0;
for ( int i = 0; i < 4; i++ )
    sum += a[i];
int avg = sum / 4;
```
2. Iterating Over Arrays

- `size_t` is a typedef for a type suitable for subscripting
- `size_t` is defined in `stdlib.h`
- Prefer `size_t` over `int` since `size_t` cannot be negative
- Example of collecting non-zero values from input array

```c
#include <stdlib.h>
#include <assert.h>

int a[] = { 0, 13, 0, 15, 17 };

size_t num_nonzeros = 0;
for ( size_t i=0; i<5; i++ )
    if ( a[i] != 0 )
        num_nonzeros++;

int b[num_nonzeros];
size_t j = 0;
for ( size_t i=0; i<5; i++ )
    if ( a[i] != 0 )
        b[j] = a[i];
    j++;

assert( num_nonzeros == j );
```
3. **Arrays as Function Parameters**

- Passing an array as a function parameter is like passing a pointer
- Arrays are *always* passed by reference
- Must pass the size along with the actual array

```c
int avg( int x[], size_t size )
{
    int sum = 0;
    for ( size_t i=0; i<size; i++ )
        sum += x[i];
    return sum / size;
}

int main( void )
{
    int a[] = { 10, 20, 30, 40 };
    int b = avg( a, 4 );
    return 0;
}
```
3. Arrays as Function Parameters

```c
void vvadd( int dest[],
            int src0[],
            int src1[],
            size_t size )
{
    for ( size_t i=0; i<size; i++ )
        dest[i] = src0[i] + src1[i];
}

int main( void )
{
    int a[] = { 1, 2, 3 };
    int b[] = { 5, 6, 7 };
    int c[] = { 0, 0, 0 };
    vvadd( c, a, b, 3 );
    return 0;
}
```
4. Multi-Dimensional Arrays

- Concept of a 1D sequence can be extended to higher dimensions
- A 2D array is just an 1D array of arrays
- Multi-dimensional arrays are stored in row-major order

```c
int matrix[3][3] = {
    { 1, 2, 3 },
    { 4, 5, 6 },
    { 7, 8, 9 },
};

int row = 2;
int col = 1;
int a = matrix[row][col];
matrix[row][col] = 13;
```

Classic matrix multiplication

```c
void mmmult( int dest[][3],
    int src0[][3],
    int src1[][3],
    size_t size )
{
    for ( int i = 0; i < size; ++i )
        for ( int j = 0; j < size; ++j )
            for ( int k = 0; k < size; ++k )
                c[i][j] += a[i][k] * b[k][j];
}
```
5. Strings

- Strings are just arrays of chars
- The length of a string is indicated in a special way
- The null terminator character (\0) indicates the end of string
- New syntax using double quotes for string literals (""")

```
1 char str[] = { 'e', 'c', 'e', '\0' };
2 char str[] = "ece";
```

- C standard library provides many string manipulation functions
- These functions are declared in the string.h header
  - strlen : calculate length of a string
  - strcmp : compare two strings
  - strcpy : copy one string to another string
  - atoi : convert a string into an integer
5. Strings

```c
int strlen( char str[] )
{
    int i = 0;
    while ( str[i] != '\0' )
        i++;
    return i;
}

int main( void )
{
    char a[] = "ece2400";
    int b = strlen( a );
    return 0;
}
```
5. Strings

```c
#include <stdio.h>
#include <string.h>

int main( void )
{
    char a[] = "ece2400";
    int b = strcmp( a, "ece2300" );
    return 0;
}
```

- Constant strings are stored in the static data section of machine memory