# Introduction to Scientific and Engineering Computation (BIL 104E)

## Lecture 9

## Arrays

# Arrays: What Is an Array?

In many cases, you have to declare a set of variables of the same data type.

Instead of declaring them individually, you can declare a set of variables of the same data type collectively as an **array**.

An **array** is a collection of variables of the same data type.

Each item in an array is called an **element**.

All **element**s in an **array** are referenced by the name of the array and are stored in a set of consecutive, adjacent memory slots.

# Arrays: Declaring Arrays

The following is the general form to declare an array:

**data-type Array-Name[Array-Size];**

**data-type** is the type specifier that indicates type of elements inside the array.

**Array-Name** is the name of the declared array.

**Array-Size** defines how many elements the array can contain.

The bracket pair ([ and ]) is also called the **array subscript operator**.

For instance:

**int array_int[8];**

**array_int** which is an array contains 8 integer **elements**.

# Arrays: Indexing Arrays

After you declare an array, you can access each of the elements in the array separately.

For instance, the following declaration generates an array of characters:

**char day[7];**

You can access the elements in the **array of day** one after another. To do this, you need a number called an **index**, enclosed in brackets, after name of the array.

All arrays in C are indexed starting at 0. In other words, the index to the first element in an array is 0, not 1. Therefore, the first element in the **array of day** is day[0]. Because there are 7 elements in the **day array**, the last element is day[6], not day[7].

# Arrays: Initializing Arrays

You can initialize each element in an array with the help of the array element.

The first way to initialized an array is to initialized one element in the array.

You can initialize the first element in the **array of day** like this:

**day[0] = ‘S’;**

Likewise, the statement

**day[1] = ‘M’;**

assigns ‘M’ to the second element, day[1], in the array.

# Arrays: Initializing Arrays

The second way to initialize an array is to initialize all elements in the array together.

For instance,

**int num_array[5] = {100, 8, 3, 365, 16};**

Here the integers inside the braces ({ and }) are correspondingly assigned to the elements of the array **num_array**.

This initialization is used instead of the first way as given below:

**num_integer[0] = 100;**
**num_integer[1] = 8;**
**num_integer[2] = 3;**
**num_integer[3] = 365;**
**num_integer[4] = 16;**

# Arrays: Initializing Arrays

Initializing an Array

```
1:  /* 12L01.c: Initializing an array */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int i;
7:     int list_int[10];
8:
9:     for (i=0; i<10; i++){
10:       list_int[i] = i + 1;
11:       printf( "list_int[%d] is initialized with %d.\n", i, list_int[i]);
12:    }
13:    return 0;
14: }
```

```
list_int[0] is initialized with 1.
list_int[1] is initialized with 2.
list_int[2] is initialized with 3.
list_int[3] is initialized with 4.
list_int[4] is initialized with 5.
list_int[5] is initialized with 6.
list_int[6] is initialized with 7.
list_int[7] is initialized with 8.
list_int[8] is initialized with 9.
list_int[9] is initialized with 10.
```

# Arrays: The Size of an Array

An array consists of consecutive memory locations.

Given an array, like this:

**data-type Array-Name[Array-Size];**

The first way to calculate the total bytes of the array is:

**sizeof(data-type) * Array-Size**

**data-type** defines the necessary number of bytes for this kind of data type.
**Array-size** defines the total elements using this number of bytes in the array.

The second way is simpler than first one and uses following expression

**sizeof(Array-Name)**

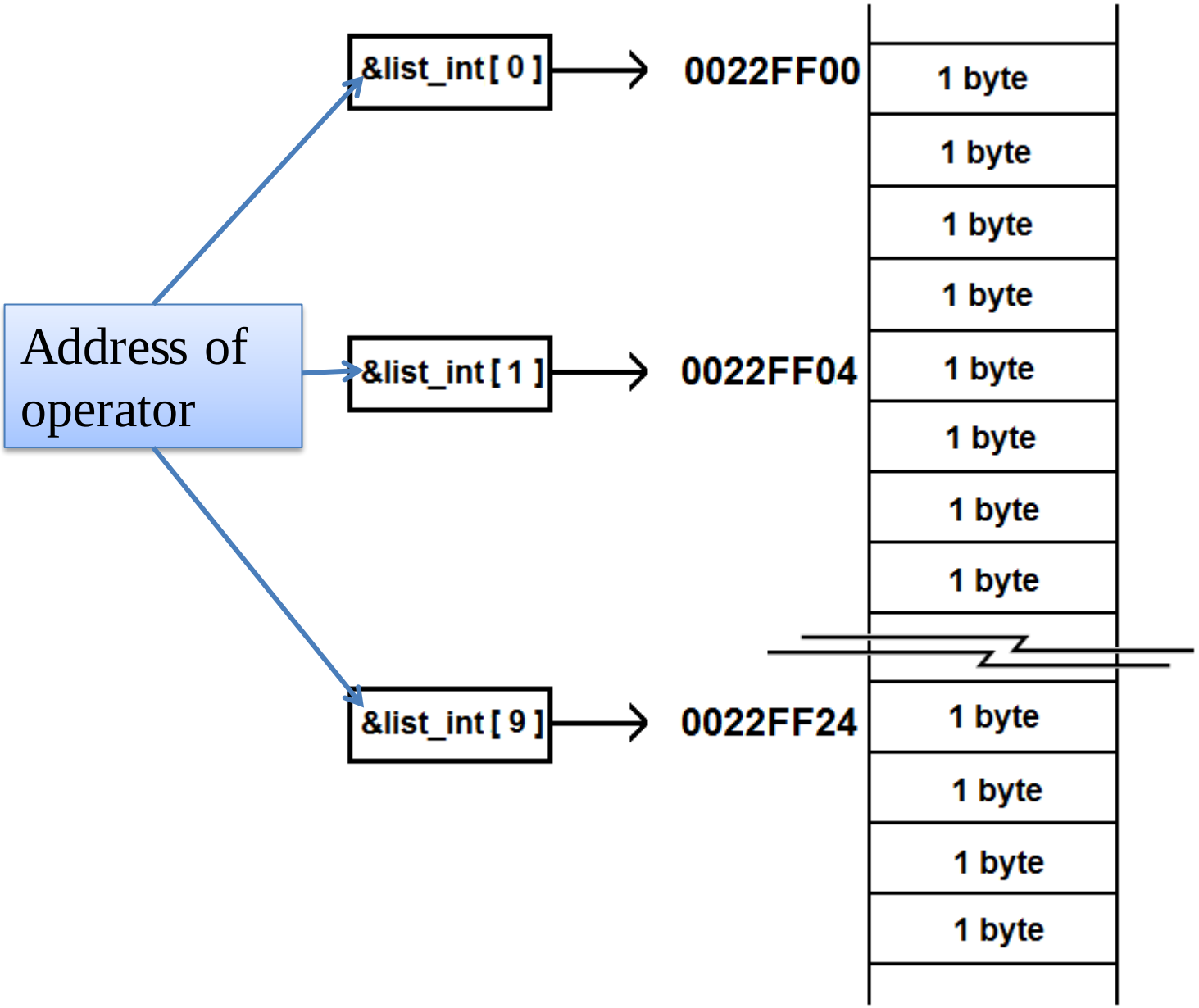
# Arrays: The Size of an Array

Calculating the Size of an Array

```
1:  /* 12L02.c: Total bytes of an array */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int total_byte;
7:     int list_int[10];
8:
9:     total_byte = sizeof (int) * 10;
10:    printf( "The size of int is %d-byte long.\n", sizeof (int));
11:    printf( "The array of 10 ints has total %d bytes.\n", total_byte);
12:    printf( "The address of the first element: %p\n", &list_int[0]);
13:    printf( "The address of the last element:  %p\n", &list_int[9]);
14:    return 0;
15: }
```

The size of int is 4-byte long.
The array of 10 ints has total 40 bytes.
The address of the first element: 0022FF00
The address of the last element: 0022FF24

# Arrays: The memory space taken by the array list_int.



# Arrays: Arrays and Pointers

You can make a pointer that refers to the first element of an array by simply assigning the array name to the pointer variable.

If an array is referenced by a pointer, the elements in the array can be accessed with the help of the pointer.

For instance, the following statements declare a pointer and an array, and assign the address of the first element to the pointer variable:

```
char   *ptr_c;
char   list_c[10];
ptr_c  =  list_c;
```

Because the address of the first element in the array **list_c** is the beginning address of the array, the pointer variable **ptr_c** is actually now referencing the array via the beginning address.

# Arrays: Increment and Decrement for Pointers

Increment and decrement operators can be used for pointers. If a pointer is increased by increment operator, it points another variable following old one.

For instance,

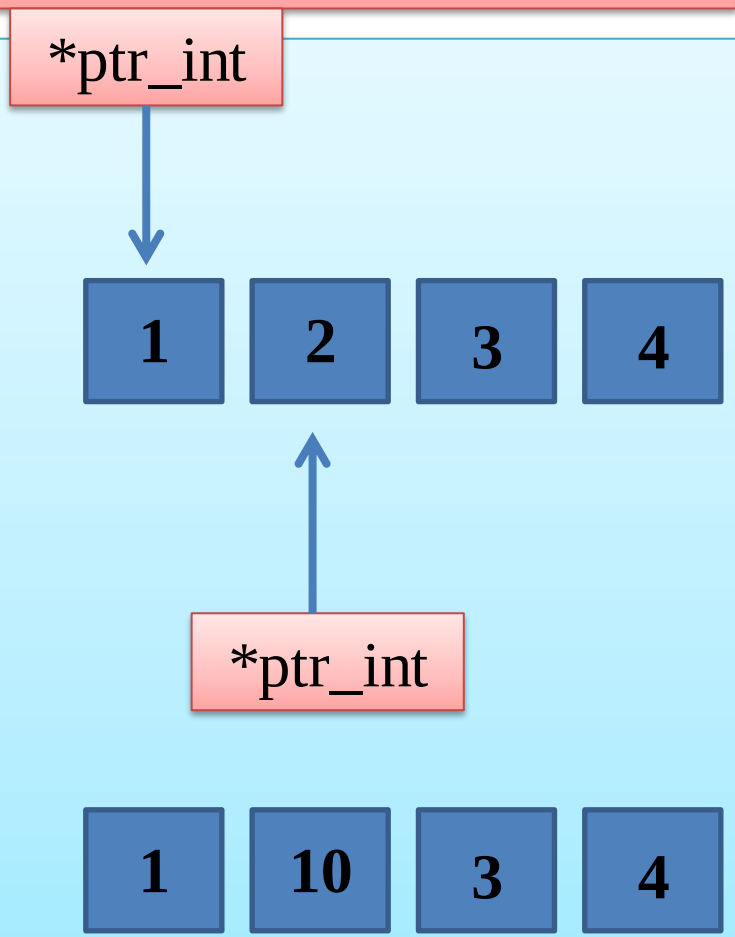
```
int   *ptr_int;

int   array_int[4] = {1, 2, 3, 4}

ptr_int = array_int;

++ptr_int;

*ptr_int = 10;
```



# Arrays: Increment and Decrement for Pointers

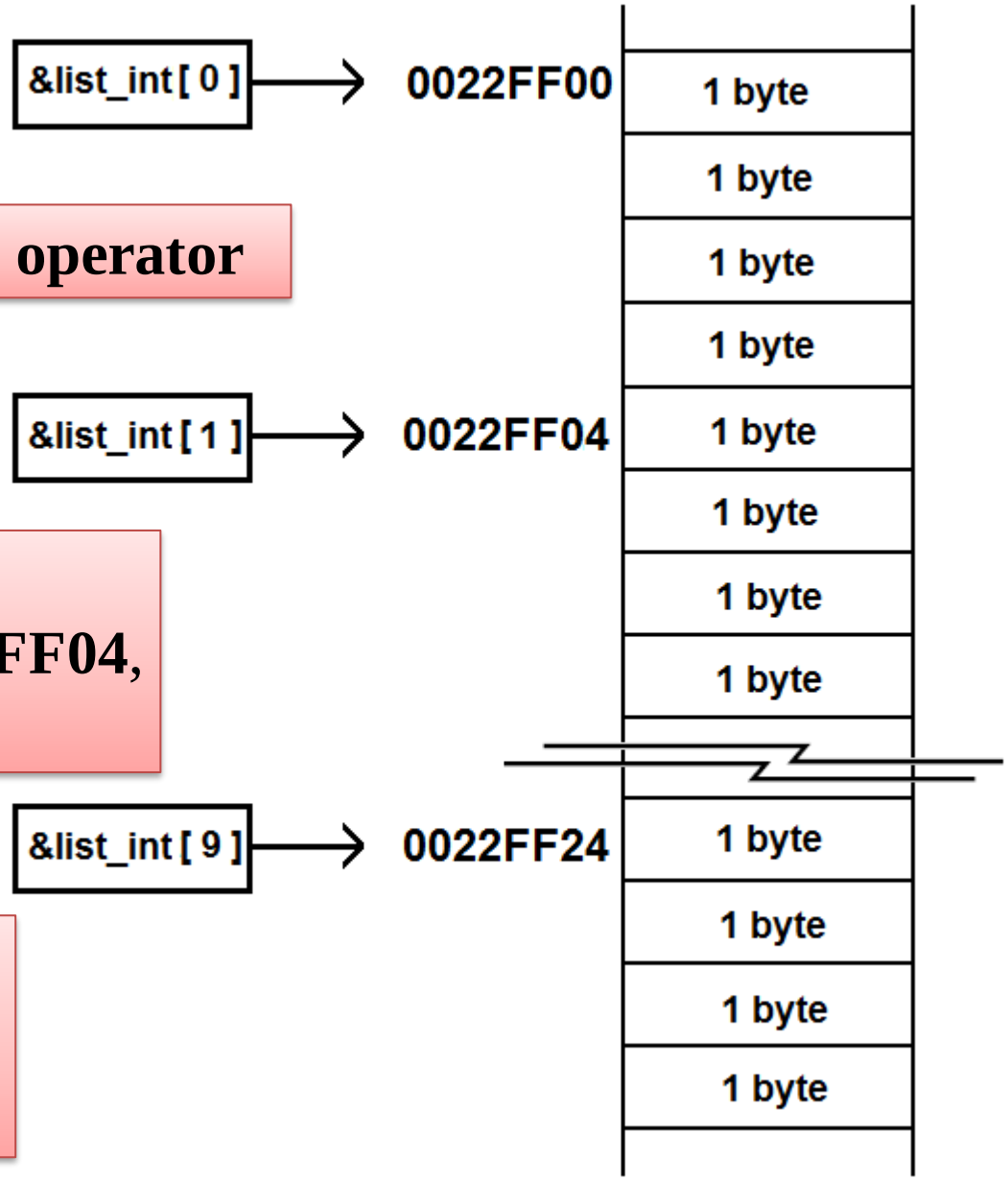
ptr_int = &list_int[0];

++ operator

++ptr_int;

*ptr_int indicates list_int[1]
Now ptr_int is equal to **0022FF04**, not **0022FF01**

Because integer type pointers can only point another integer type variable.



# Arrays: Arrays and Pointers

Referencing an Array with a Pointer

```
1:  /* 12L03.c: Referencing an array with a pointer */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int *ptr_int;
7:     int list_int[10];
8:     int i;
9:
10:    for (i=0; i<10; i++)
11:       list_int[i] = i + 1;
12:    ptr_int = list_int;
13:    printf( "The start address of the array: %p\n", ptr_int);
14:    printf( "The value of the first element: %d\n", *ptr_int);
15:    ptr_int = &list_int[0];
16:    printf( "The address of the first element: %p\n", ptr_int);
17:    printf( "The value of the first element: %d\n", *ptr_int);
18:    return 0;
19: }
```

```
The start address of the array: 0022FF00
The value of the first element: 1
The address of the first element: 0022FF00
The value of the first element: 1
```

# Arrays: Displaying Arrays of Characters

A character string is defined as a contiguous sequence of characters terminated by the null character ('\0').

Printing an Array of Characters

```
1:  /* 12L04.c: Printing out an array of characters */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char array_ch[7] = {'H', 'e', 'l', 'l', 'o', '!', '\0'};
7:     int i;
8:
9:     for (i=0; i<7; i++)
10:       printf("array_ch[%d] contains: %c\n", i, array_ch[i]);
11:    /*---  method I ---*/
12:    printf( "Put all elements together(Method I):\n");
13:    for (i=0; array_ch[i] != '\0' && i<7; i++)
14:       printf("%c", array_ch[i]);
15:    /*---  method II ---*/
16:    printf( "\nPut all elements together(Method II):\n");
17:    printf( "%s\n", array_ch);
18:
19:    return 0;
20: }
```

# Arrays: Displaying Arrays of Characters

## Computer Screen

```
array_ch[0] contains: H
array_ch[1] contains: e
array_ch[2] contains: l
array_ch[3] contains: l
array_ch[4] contains: o
array_ch[5] contains: !
array_ch[6] contains:
Put all elements together(Method I):
Hello!
Put all elements together(Method II):
Hello!
```

# Arrays: The Null Character (‘\0’)

The null character (‘\0’) is treated as one character in C.

It is a special character that marks the end of a string.

For instance,
When functions like **printf()** act on a character string, they process one character after another until they encounter the **null character (‘\0’)** .

The null character (‘\0’), which is always evaluated as a value of zero, can also be used for a logical test in a control-flow statement.

# Arrays: The Null Character ('\0')

Ending Output at the Null Character

```
1:  /* 12L05.c: Stopping at the null character */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char array_ch[15] = {'C', ' ',
7:                          'i', 's', ' ',
8:                          'p', 'o', 'w', 'e', 'r',
9:                          'f', 'u', 'l', '!', '\0'};
10:    int i;
11:    /*  array_ch[i] in logical test */
12:    for (i=0; array_ch[i]; i++)
13:       printf("%c", array_ch[i]);
14:
15:    printf("\n");
16:    return 0;
17: }
```

```
C is powerful!
```

# Arrays: Multidimensional Arrays

So far, all the arrays have been **one-dimensional arrays**.

In addition to **one-dimensional arrays**, the C language also supports **multidimensional arrays**.

The general form of declaring a N-dimensional array is

**data-type Array-Name[Array-Size1][Array-Size2]. . . [Array-SizeN];**

where **N** can be any positive integer.

Because the two-dimensional array is widely used, let's focus on two-dimensional arrays

For instance,

**int array_int[2][3];**

This statement declares a two dimensional integer array

# Arrays: Multidimensional Arrays

You can initialize the two-dimensional array **array_int** in the following ways:

First way:

```
array_int[0][0] = 1;
array_int[0][1] = 2;
array_int[0][2] = 3;
array_int[1][0] = 4;
array_int[1][1] = 5;
array_int[1][2] = 6;
```

Second way:

```
int array_int[2][3] = {1, 2, 3, 4, 5, 6};
```

Third way:

```
int array_int[2][3] = { {1, 2, 3}, {4, 5, 6} };
```

# Arrays: Multidimensional Arrays

## Printing a Two-Dimensional Array

```
1:  /* 12L06.c: Printing out a 2-D array */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     int two_dim[3][5] = {1, 2, 3, 4, 5,
7:                          10, 20, 30, 40, 50,
8:                          100, 200, 300, 400, 500};
9:     int i, j;
10:
11:    for (i=0; i<3; i++){
12:       printf("\n");
13:       for (j=0; j<5; j++)
14:          printf("%6d", two_dim[i][j]);
15:    }
16:    printf("\n");
17:    return 0;
18: }
```

```
     1     2     3     4     5
    10    20    30    40    50
   100   200   300   400   500
```

# Arrays: Unsized Arrays

The size of a dimension is normally given during the declaration of an array.

However C compiler can actually calculate a dimension size of an array automatically if an array is declared as an unsized array.

For example,
when the compiler sees the following unsized array:

```
int    list_int[ ] = { 10, 20, 30, 40, 50, 60, 70, 80, 90};
```

it will create an array big enough to store all the elements.

You can declare a multidimensional unsized array like this:

```
char list_ch[ ][2] = {
                ‘a’, ‘A’,
                ‘b’, ‘B’,
                ‘c’, ‘C’,};
```

# Arrays: Unsized Arrays

## Initializing Unsized Arrays

```
1:  /* 12L07.c: Initializing unsized arrays */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char array_ch[] = {'C', ' ',
7:                        'i', 's', ' ',
8:                        'p', 'o', 'w', 'e', 'r',
9:                        'f', 'u', 'l', '!', '\0'};
10:    int list_int[][3] = {
11:           1, 1, 1,
12:           2, 2, 8,
13:           3, 9, 27,
14:           4, 16, 64,
15:           5, 25, 125,
16:           6, 36, 216,
17:           7, 49, 343};
18:
19:    printf("The size of array_ch[] is %d bytes.\n", sizeof (array_ch));
20:    printf("The size of list_int[][3] is %d bytes.\n", sizeof (list_int));
21:    return 0;
22: }
```

# Arrays: Unsized Arrays

## Computer Screen

```
The size of array_ch[] is 15 bytes.
The size of list_int[][3] is 84 bytes.
```