Procedure Programming

Arrays

Instructor: Jeeho Ryoo

Announcement

- Lab1 remarks, which I hate to talk about
  - Why not read lab PDF?
  - Why think when you don't have good background knowledge?
- Lab 1 remarks, which I love to talk about
  - Great teamwork and help
  - People looking things up
  - Educated thoughts and questions
- Why I am who I am
  - 3 types of students in my world
  - 2 types of people in my world
- One teaching belief that I always carry
- Dev and testing platform

Arrays

Collections or groups of the same data type

Can be multi-dimensional (arrays of arrays)

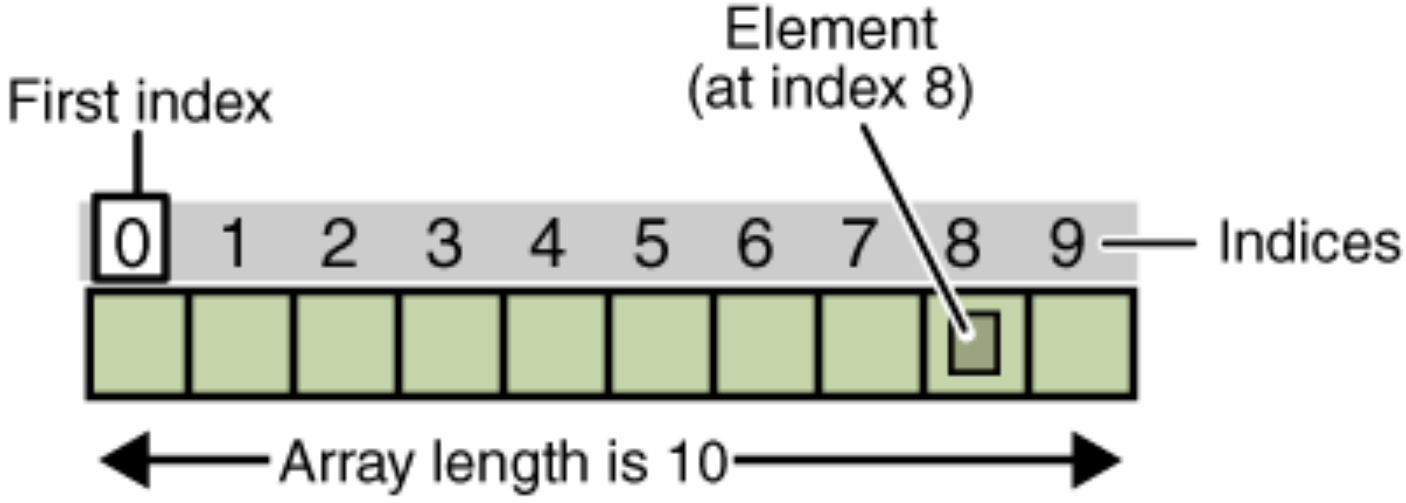
- 1D, 2D, 3D

Variable name is followed by one pair of square brackets

- Ex: Regular integer variable declaration: *int x;*
- Array of integers variable declaration: *int x[];*

Arrays

- Arrays are referenced using indices
- Indices of arrays always start at 0



Declaring an Array

The length of the array can be declared by putting the length in the square bracket []

int x[3];

We can load the array with values when we declare it as well

int x[5] = {4, 6, 7};

Declaring an Array

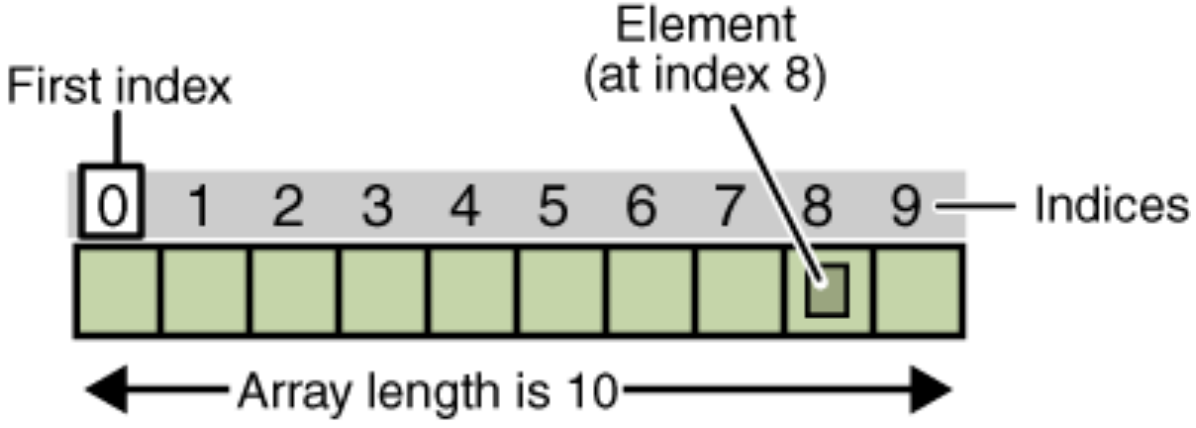
We can declare and load the array with values without setting it's length

int y[] = {4, 5, 3, 8, 9, 12, 45};

Accessing elements in an Array

We access elements in the array by using an index number
In this example we assign the 8th element of the array to *y*

```
int x[10] = {4, 6, 8, 6, 55, 6, 8, 10, 4, 0};
int y = x[8];
```



Accessing elements in an Array (Example)

Write a program that prints the 6th element of an Array

```
#include <stdio.h>

int main() {
    int z[] = {6, 5, 8, 10, 67, 43, 200};
    printf ("%d", z[5]);
    return 0;
}
```

Using Arrays (Example 1)

What will be displayed in console?

```
#include <stdio.h>

int main() {
    int x[7] = {0, 1, 5, 2, 3, 8, 6};
    int y = x[2]*x[4];
    printf ("%d", y);
    return 0;
}
```

Using Arrays (Example 2)

What will z have to equal so 8 is displayed in console?

```
#include <stdio.h>

int main()  {
    int x[] = {9, 4, 1, 8, 34, 6, 8, 56, 87};
    int z = ?;
    int y = z – (x[2]*x[4]);
    printf (“%d”, y);
    return 0;
}
```

2d Arrays

2d arrays are also known as an array of 1d arrays
Declared by providing more then one set of square brackets

int x[3][3];

2d Arrays (Cont.)

Typically the first dimension is the number of rows and second dimension is the number of columns
Visualizing: *int x[3][3];*

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Declaring 2d Arrays

Typically you declare and load a 2d array by grouping the { }

int x[2][2] = { {4, 5}, {8, 2} }; or
int x[2][2] = {4, 5 ,8 ,2 };

2d arrays can also be loaded without providing a length for the first square bracket []

int x[][2] = { {4, 5}, {8, 2} };

Declaring 2d arrays (cont.)

Declaration needs to be loaded if first square bracket is empty

int x[][2];

int x[][2] = { {4, 5}, {8, 2} };

Accessing elements in a 2d array

Elements of 2d arrays are accessed using the index number of the row or column
In this example we give the 1st row of the 2nd column a value of 8.

int x[2][2];
x[0][1] = 8;

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Accessing elements in a 2d Array (Example)

Write a program that prints the 3rd element of the 2nd row

```
#include <stdio.h>
int main() {
    int z[][3] = { {6, 5, 8}, {10, 6, 3}, {5, 7, 1} };
    printf ("%d", z[1][2]);

    return 0;
}
```

Using 2d arrays (Example 1)

What will be displayed in console?

```
#include <stdio.h>

int main()  {
    int x[4][2] = { {4, 7}, {2, 8}, {8, 4}, {9, 0} };
    printf ("%d", x[0][0]);
    return 0;
}
```

Using 2d arrays (Example 2)

What will be displayed in console?

```
#include <stdio.h>
int main() {
    int x[][4] = { {0, 1, 5, 7 }, {3, 8, 6, 9} };
    int y = x[0][1] * x[1][3];
    printf (“%d”, y);
    return 0;
}
```

Determining the size of an array

Best way to get the length of an array is to use the *sizeof* operator

sizeof operator returns the size in bytes

Example:

```
int x[2];
int y = sizeof(x);
```

Y equals ‘8’, because the array is 8 bytes in size

Determining the length of an array

Each element in array is equal in size because they are same types (int, float, char, etc)
Note the value can be different, but size will be the same

You can divide the array size by the size of just one element

```
int x[2];
int y = sizeof(x) / sizeof(x[0]);
```

Returns 2, which is the length of the array

Bounds checking with arrays

In other languages bounds checking is better handled then in C
In C you can easily create issues without realizing it

Consider this 2d array:

int x[3][3] = { {2, 3, 6}, {2, 6, 6}, {2, 6, 8, 7} };

The 7 is outside the declared scope of the array

Bounds checking with arrays (cont.)

Accessing elements outside the array bounds can also cause issues

Consider this:

```
int x[3][3] = { {2, 3, 8}, {2, 6, 6}, {2, 6, 8} };
int y = x[2][50];
```

The arrays length does not support this index
Could be some other value you are accessing

Multi-dimensional arrays extra

int x[2][2] = { {4, 5}, {6, 7} };

To get the number 7 in the second row we normally write:

x[1][1]; (recommended)

This will work to:

x[0][3]; (Not recommended)

This is because of the assigned memory for the array

Examples

`int A[10]`

- An array of ten integers
- **`A[0]`**, **`A[1]`**, …, **`A[9]`**

`double B[20]`

- An array of twenty long floating point numbers
- **`B[0]`**, **`B[1]`**, …, **`B[19]`**

Arrays of **`structs`**, **`unions`**, **`pointers`**, etc., are also allowed

Array indexes *always* start at zero in *C*

Examples

`int C[]`

- An array of an unknown number of integers (allowable in a parameter of a function)
- **`C[0]`**, **`C[1]`**, …, **`C[`***max-1*`]`**

`int D[10][20]`

- An array of ten rows, each of which is an array of twenty integers
- **`D[0][0]`**, **`D[0][1]`**, …, **`D[1][0]`**, **`D[1][1]`**, …, **`D[9][19]`**
- *Not used so often as arrays of pointers*

Two-dimensional Arrays

`int D[10][20]`

- A *one-dimensional array* with 10 elements, each of which is an array with 20 elements

i.e., **`int D[10][20]  /*[row][col]*/`**

Last subscript varies the fastest

- I.e., elements of last subscript are stored contiguously in memory

Also, three or more dimensions

Array Elements (continued)

Array elements are commonly used in loops
E.g.,

```
for(i=0; i < max; i++)
        A[i] = i*i;

for(sum = 0, j=0; j < max; j++)
        sum += B[j];

for(count=0; rc!=EOF; count++)
        rc=scanf("%f", &A[count]);
```

Caution! Caution! Caution!

It is the programmer's responsibility to avoid indexing off the end of an array

- *Likely* to corrupt data
- May cause a *segmentation fault*
- Could expose system to a *security hole!*

C does NOT check *array bounds*

- I.e., whether index points to an element within the array
- Might be high (beyond the end) or negative (before the array starts)

Declaring Arrays

Static or automatic

Array size may be determined explicitly or implicitly

Array size may be determined at run-time

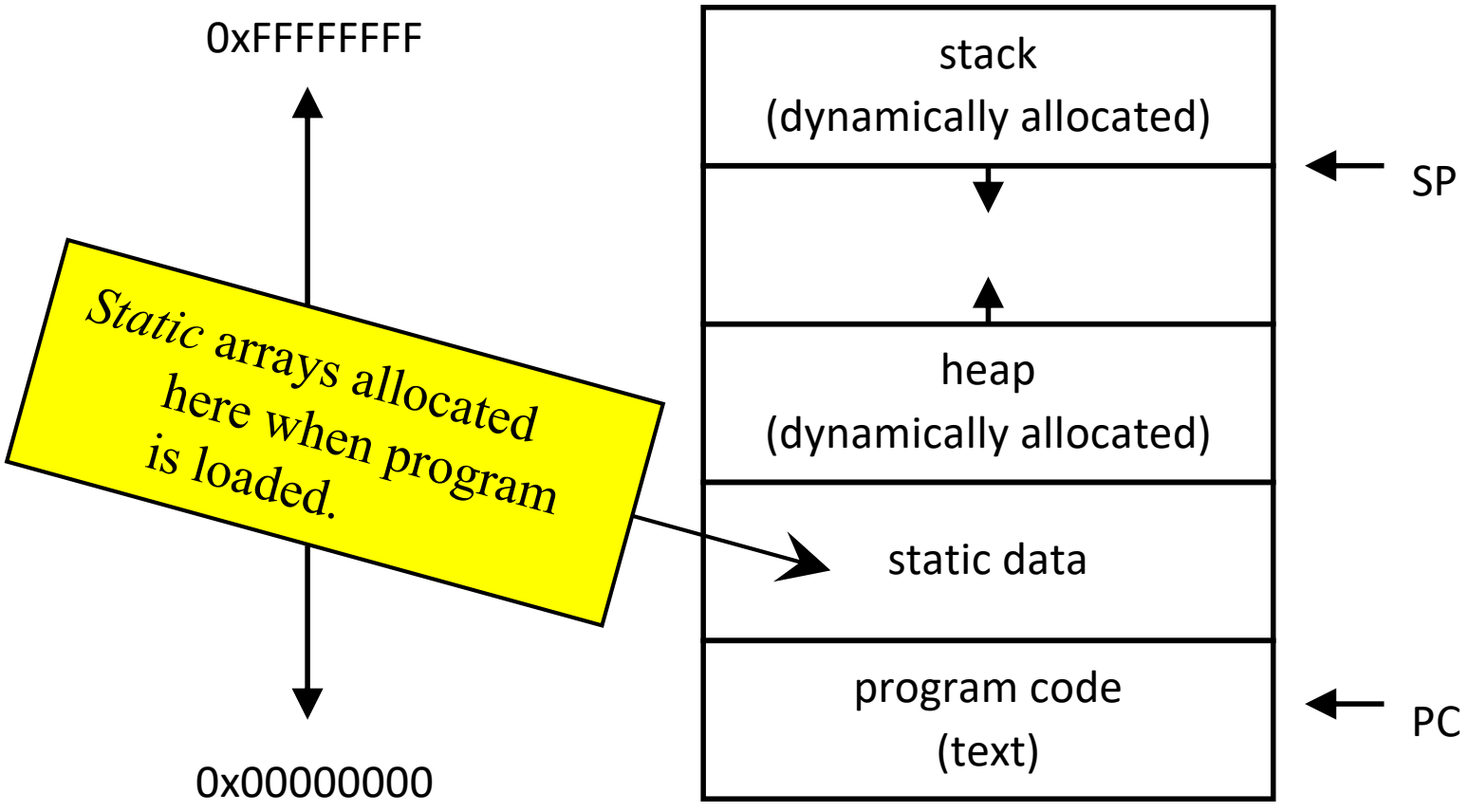
Declaring Arrays (continued)

Outside of any function – always static

```
int A[13];

#define CLASS_SIZE 73
double B[CLASS_SIZE];

int nElements = 25
float C[nElements];

static char D[256];
```

Static ⇒ retains values across function calls

Static Data Allocation

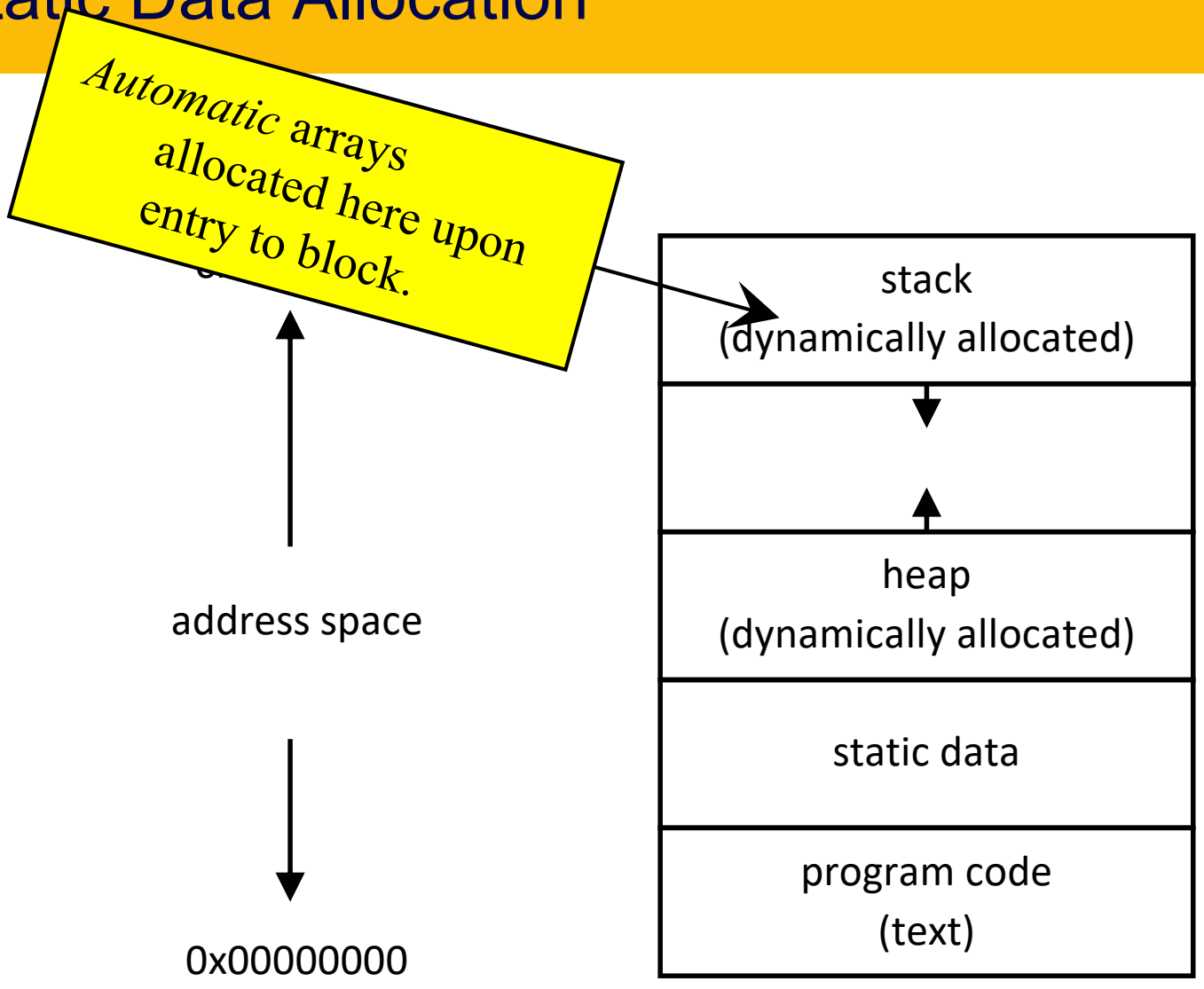


Declaring Arrays (continued)

Inside function or compound statement – usually automatic

```
void f( …) {
     int A[13];

     #define CLASS_SIZE 73
     double B[CLASS_SIZE];

     int nElements = 25
     float C[nElements];

     static char D[256]; /*static, not
     visible outside function */

}    //f
```

Static Data Allocation



Implicit Array Size Determination

```
int days[] = {31, 28, 31, 30, 31, 30, 31,
31, 30, 31, 30, 31};
```

Array is created with as many elements as initial values

In this case, 12 elements

Values must be compile-time constants (for static arrays)

Values may be run-time expressions (for automatic arrays)

Getting Size of Implicit Array

`sizeof` operator – returns # of bytes of memory required by operand

Examples:–

- **`sizeof (int)`** – # of bytes per **`int`**
- **`sizeof (float)`** – # of bytes per **`float`**

Must be able to be determined at compile time

- Getting size of dynamically allocated arrays not supported

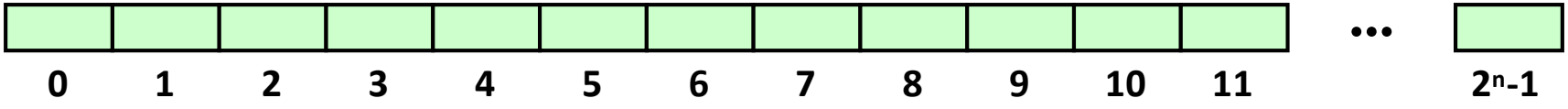
Initializing a Two-Dimensional Array

```
static char daytab[2][12] = {
    {31,28,31,30,31,30,31,31,30,31,30,31},
    {31,29,31,30,31,30,31,31,30,31,30,31}
};//daytab
```

OR

```
static char daytab[2][12] = {
    31,28,31,30,31,30,31,31,30,31,30,31,
    31,29,31,30,31,30,31,31,30,31,30,31
};//daytab
```

Digression – Memory Organization



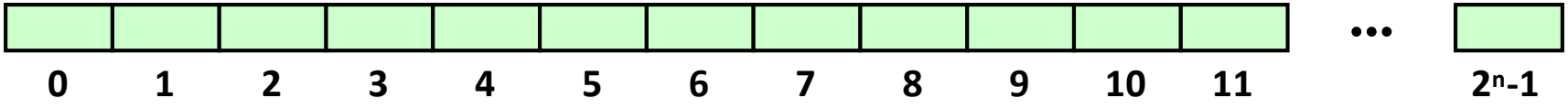
All modern processors have memories organized as sequence of *numbered bytes*

- Many (but not all) are linear sequences

Definitions:–

- *Byte:* an 8-bit memory cell capable of storing a value in range 0 … 255
- *Address:* number by which a memory cell is identified

Memory Organization (continued)



Larger data types are sequences of bytes – e.g.,

- **`short int`** – 2 bytes
- **`int`** – 2 or 4 bytes
- **`long`** – 4 or 8 bytes
- **`float`** – 4 bytes
- **`double`** – 8 bytes

(Almost) always aligned to multiple of size in bytes

Address is "first" byte of sequence (i.e., byte zero)

- May be low-order or high-order byte
- *Big endian* or *Little endian*