Arrays in C

ITCS 2116: C Programming

College of Computing and Informatics

Department of Computer Science

Motivation to Use Arrays?

- Simple data type: variables of these types can store only one value at a time
- Structured data type: a data type in which each data item is a collection of other data items. Arrays are a structured data type.

Arrays

- A **collection** of a **fixed number** of components, all of the **same data type**
- One-dimensional array: components are arranged in a list form
- Syntax for declaring a one-dimensional array:

```
dataType arrayName[intExp];
```

- **`intExp`**: any **constant expression** that evaluates to a positive integer

Declaring Arrays

- The declaration determines the
  1. element **datatype**
  2. array **length** (implicit or explicit)
  3. array **initialization** (none, partial, or full)
- Array length (*bounds*) can be any constant (integer) expression, e.g., `3`, `3*16-20/4`, etc.

Accessing Array Components

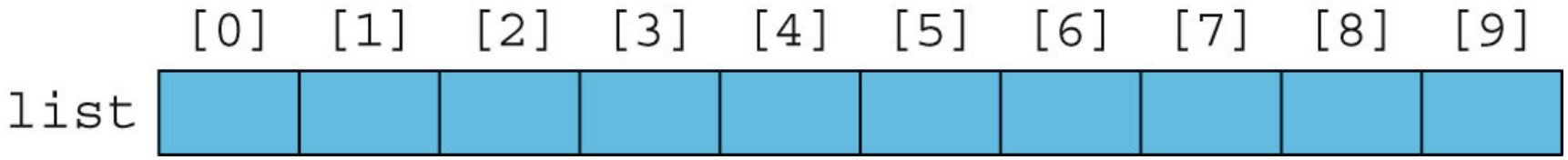
- General syntax:

```
arrayName[indexExp]
```

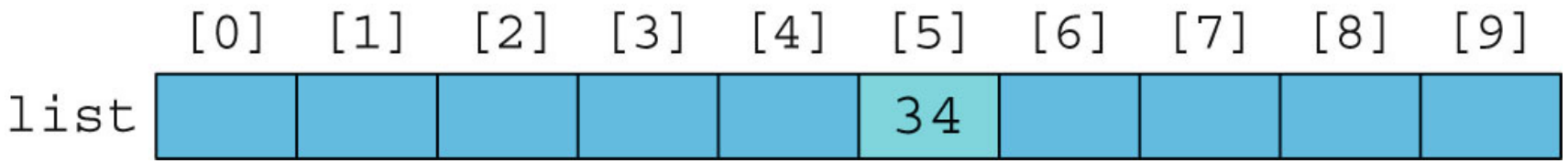
- `indexExp`: called the **index**
  - An expression with a nonnegative integer value
- Value of the index is the position of the item in the array
- **`[]`** : array subscripting operator
  - Array index always starts at 0

Accessing Array Components (cont'd.)

```
int list[10];
```

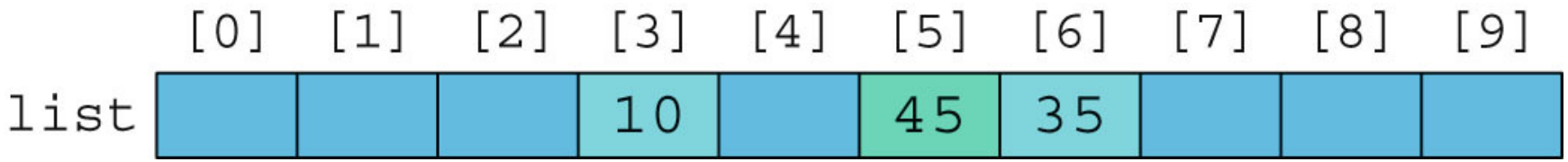



```
list[5] = 34;
```



Accessing Array Components (cont'd.)

```
list[3] = 10;
list[6] = 35;
list[5] = list[3] + list[6];
```



Processing One-Dimensional Arrays

- Basic operations on a one-dimensional array:
  - Initializing
  - Inputting data
  - Outputting data stored in an array
  - Finding the largest and/or smallest element
- Each operation requires ability to step through elements of the array
  - Easily accomplished by a **loop**

Arrays

- Almost any interesting program uses **for loops** and **arrays**
- **`a[i]`** refers to **`i`**th element of array **`a`**
  - **numbering starts at 0**

Processing One-Dimensional Arrays (cont'd.)

```
int list[5];  //array of size 5
int i;

for (i = 0; i < 5; i++)
{
   scanf("%d", &list[i]);
}

for (i = 0; i < 5; i++)
{
   printf("%d\n", list[i]);
}
```

Array Initialization

Initializing 1-D Arrays

- Explicit length, nothing initialized:

```
int   days_in_month[12];
char  first_initial[12];
float inches_rain[12];
```

- Explicit length, **fully** initialized:

```
int days_in_month[12]
= {31,28,31,30,31,30,31,31,30,31,30,31 };

char first_initial[12]
= {'J','F','M','A','M','J','J','A','S','O','N','D'};

float inches_rain[12]
= {3.5,3.7,3.8,2.6,3.9,3.7,4.0,4.0,3.2,2.9,3.0,3.2};
```

What happens if you try to initialize more than 12 values??

Initializing 1-D Arrays (cont'd)

- **Implicit** length + **full** initialization:

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

char first_initial[]
= {'J','F','M','A','M','J','J','A','S','O','N','D'};

float inches_rain[]
= {3.5,3.7,3.8,2.6,3.9,3.7,4.0,4.0,3.2,2.9,3.0,3.2};
```

The number of values initialized implies the size of the array.

Initializing 1-D Arrays (cont'd)

- Can initialize just **selected** elements
  - uninitialized values are cleared to **0**

- Two styles:

```
int days_in_month[12]
= {31,28,31,30,31,30};

char first_initial[12]
= {'J','F','M'};

float inches_rain[12]
= {3.5,3.7,3.8,2.6,3.9,3.7,4.0,4.0};
```

```
int days_in_month[12]
= {[0]=31,[3]=30,[7]=31};

char first_initial[12]
= {[2]='M',[3]='A', [4]='M', [11]='D'};
```

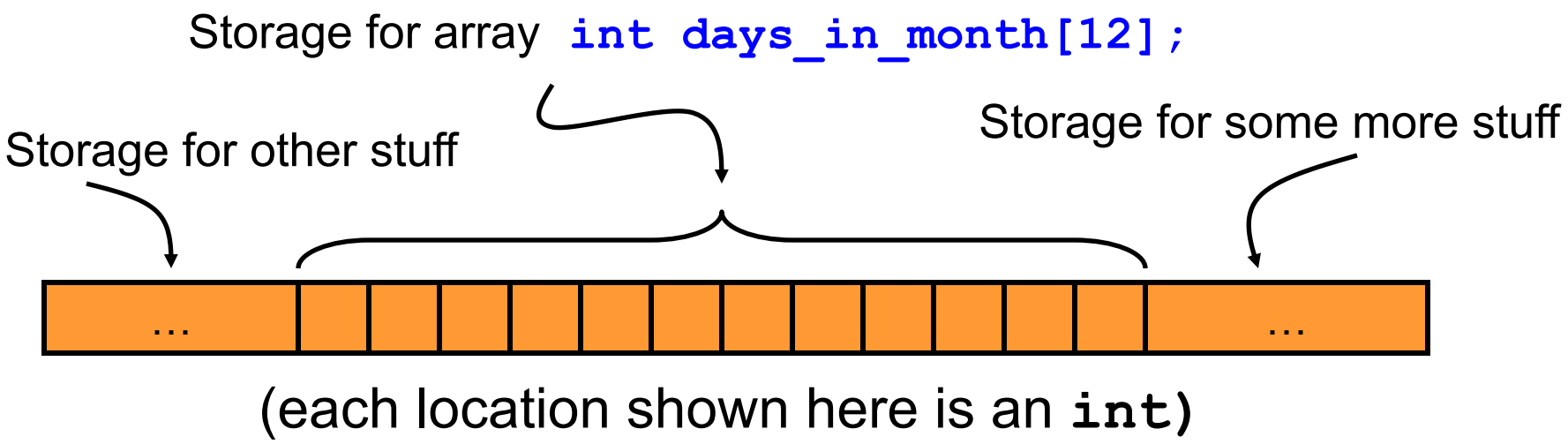
Initializing 1-D Arrays (cont'd)

Implicit array length **and partial** initialization??

```
char first_initial[] =
    { [0]='J', [2]='M', [8]='S' };
```

How big is this array?

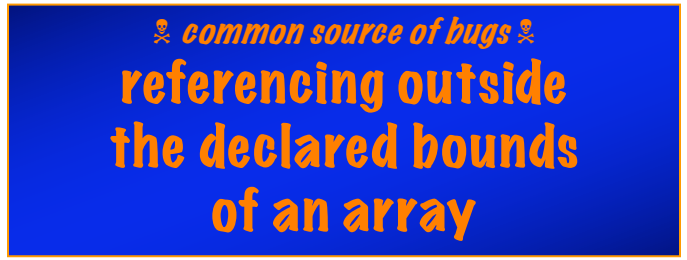
Memory Layout and Bounds Checking



- There is **NO bounds checking** in C
  - i.e., it's legal (but not advisable) to refer to `days_in_month[216]` **or** `days_in_month[-35]` !
  - Who knows what is stored there?

Bounds Checking… (cont'd)

- References outside of declared array bounds
  - may cause program exceptions ("**bus error**" or "**segmentation fault**"),
  - may cause other data values to become corrupted, or
  - may just reference wrong values
- Debugging these kinds of errors is one of the hardest errors to diagnose in C



Operations on Arrays

- The only **built-in operations on arrays** are:
  - address of operator (`&`)
  - `sizeof` operator
  - *we'll discuss these shortly…*
- Specifically, there are **no operators** to…
  - assign a value to an entire array
  - add two arrays
  - multiply two arrays
  - rearrange (permute) contents of an array
  - etc.

Operations on Arrays?

Instead of using built-in operators, write **loops** to process arrays.
For example:

```
int exam1_grade[NUMSTUDENTS],
    hw1[NUMSTUDENTS],
    hw2[NUMSTUDENTS],
    hwtotal[NUMSTUDENTS];

for (int j = 0; j < NUMSTUDENTS; j++) {
    exam1_grade[j] = 100;
    hwtotal[j] = hw1[j] + hw2[j];
}
```

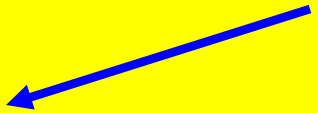
Variable Length Arrays

In C99, array length can be **dynamically** declared for non-static variables:

```
int i, szar;

(void) printf("Enter # of months in year: ");
(void) scanf("%d", &szar);

int days[szar];
```



What happens if you attempt to allocate an array of size zero, or of negative size??

Variable… (cont'd)

However… array lengths cannot change dynamically during program execution

```
int sz1, sz2;
(void) printf("Enter first # of records: ");
(void) scanf("%d", &sz1);
int recs[sz1];

… do some stuff…

(void) printf("Enter second # of records: ");
(void) scanf("%d", &sz2);
int recs[sz2];
```

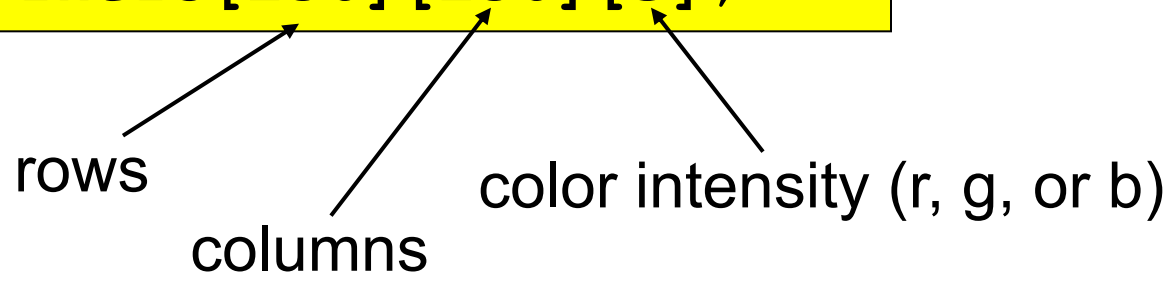
Will not work! Compile error!

Multidimensional Arrays

Multi-Dimensional ("M-D") Arrays

Declaring a multi-dimensional array with **explicit** length (in all dimensions), **no** initialization:

```
int xy_array[10][20];
char rgb_pixels[256][256][3];
```



Referring to one element of a multi-dimensional array:

```
xyval = xy_array[5][3];
r = rgb_pixels[100][25][0];
```

M-D Arrays… (cont'd)

- M-D Arrays are really **arrays of arrays**! i.e.,
  - 2-D arrays (**`xy_array`**) are arrays of 1-D arrays
  - 3-D arrays (**`rgb_pixels`**) are arrays of 2-D arrays, each of which is an array of 1-D arrays
  - etc.
- The following are **all** valid references

```
rgb_pixels              /* entire array (image)
                                of pixels  */
rgb_pixels[9]           /* 10th row of pixels   */
rgb_pixels[9][4]        /* 5th pixel in 10th row */
rgb_pixels[9][4][0]     /* red value of 5th
                           pixel in 10th row    */
```

Initializing M-D Arrays

With **implicit** initialization, elements are initialized in "leftmost-to-rightmost" dimension order, e.g.

```
/* 2-D array with 2 rows and 3 columns */
char s2D[2][3] =
     { {'a', 'b', 'c'}, {'d', 'e', 'f'} };

for (int i = 0; i < 2; i++)
     for (int j = 0; j < 3; j++)
          putchar(s2D[i][j]);
```

The above outputs **abcdef**

Initializing M-D… (cont'd)

Full initialization, explicit length

```
int i[3][4] =
{ {0, 1, 2, 3},
  {4, 5, 6, 7},
  {8, 9, 10, 11} };
```

Partial initialization, explicit length

```
int i[3][4] =
{ {0, 1},
  {4, 5},
  {8, 9} };
```

Implicit Length for M-D Arrays

Only the first dimension (row) length can be omitted

OK

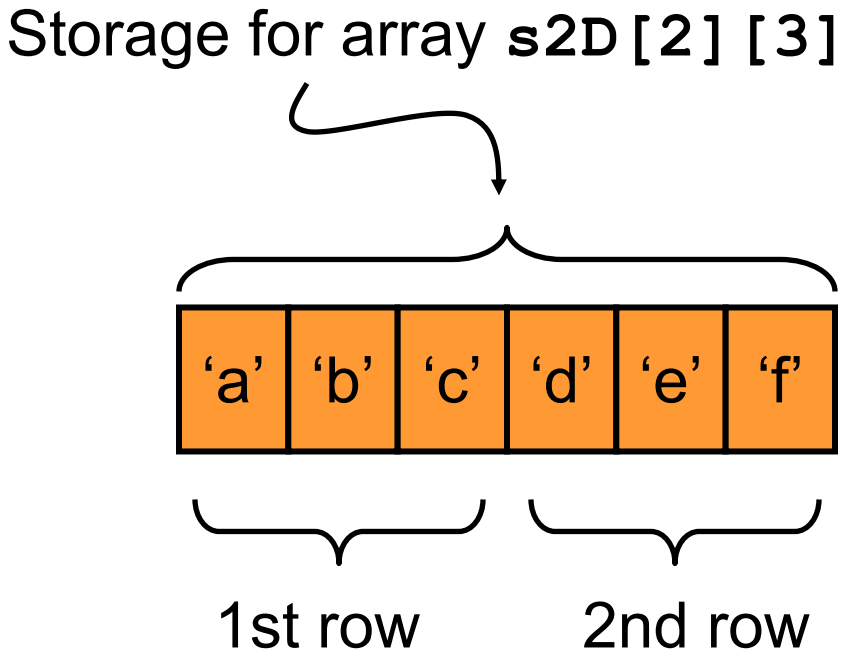
```
int i[][3] =
{ {0, 1, 2}, {4, 5, 6} };
```

NOT OK

```
int i[2][] =
{ {0, 1, 2}, {4, 5, 6} };
```

Memory Layout of M-D Arrays

Laid out in **row-major** (leftmost-to-rightmost dimension) ordering



Doesn't matter what the order is, in Java; why should we care in C?

Character Strings

Character Strings

- **Strings** (i.e., sequence of **`chars`**) are a particularly useful 1-D array
- All the rules of arrays apply, but there are a couple of extra features
- Initialization can be done in the following styles

```
char s1[] = "hope";
char s2[] = { 'h', 'o', 'p', 'e' };
```

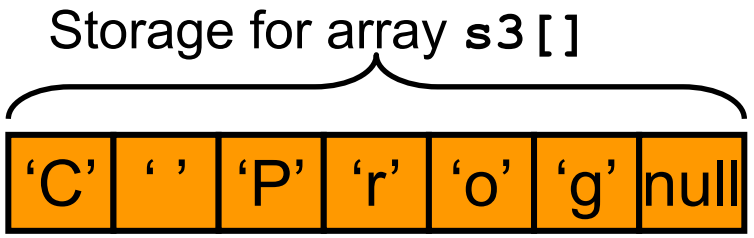
☠ *common source of bugs* ☠
failure to null terminate a string

- In the first style, the string is implicitly null-terminated by the compiler, i.e., the array is 5 characters long

Character Strings (cont'd)

- Null termination is a convenience to avoid the need to specify explicitly the length of a string
  - i.e., functions processing strings can check for a null character to recognize the end of the string
  - For example, **`printf()`** displays a string of arbitrary length using format specifier **`%s`** (the string must be null-terminated)

```
char s3[] = "C Prog";
printf ("The string is %s\n", s1);
```



Each location shown here is a **`char`**

Character String Concatenation

- Can initialize a string as a concatenation of multiple quoted initializers:

```
char s1[] = "Now " "is " "the " "time";
printf("%s\n", s1);
```

Output of execution is:

```
Now is the time
```

- Can use anywhere a string constant is allowed

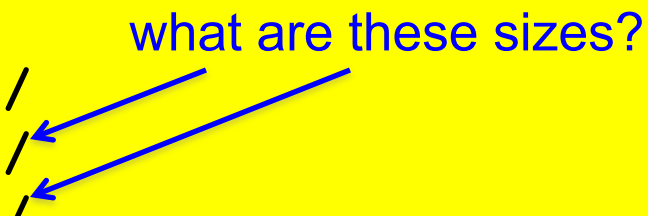
```
char s1[] = "This is a really long string that"
            "would be hard to specify in a single"
            "line, so using concatenation is a"
            "convenience." ;
```

The `sizeof` Operator

- Not a function call; a C operator
  - Returns number of bytes required by a data type
- Return value is of predefined type `size_t`

```
#include <stdlib.h>
size_t tsz1, tsz2, tsz3;
int a;
float b[100];
struct student { …definition here… } st;

tsz1 = sizeof (a);  /* 4 */
tsz2 = sizeof (b);  /* ? */
tsz3 = sizeof (st); /* ? */
```



The `sizeof` Operator (cont'd)

Can also be used to determine the number of elements in an array

```
float b[100];
…
int nelems;
nelems = sizeof (b) / sizeof (b[0]);
```

`sizeof()` is evaluated at compile time for statically allocated objects

Arrays

- Specification of array and index is *commutative*, i.e., `a[i]` references the same value as `i[a]`

```
days_in_month[0] = 31;
1[days_in_month] = 28;
```

- The syntax used on the second line is not very common and it is **not** recommended.

References

- K. N. King, *C Programming: A Modern Approach*, 2nd Edition. W. W. Norton & Company. 2008.

- D.S. Malik, *C++ Programming: From Problem Analysis to Program Design*, Seventh Edition. Cengage Learning. 2014.