Arrays in C

ITSC 2181: Introduction to Computer Systems
UNC Charlotte
College of Computing and Informatics

COLLEGE OF COMPUTING
AND INFORMATICS

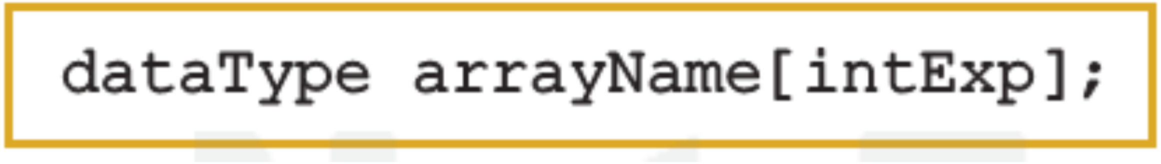
Motivation to Use Arrays?

- 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.

COLLEGE OF COMPUTING
AND INFORMATICS

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];
```

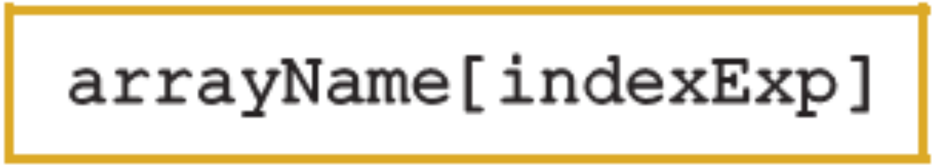
- **`intExp`**: any **constant expression** that evaluates to a positive integer

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:

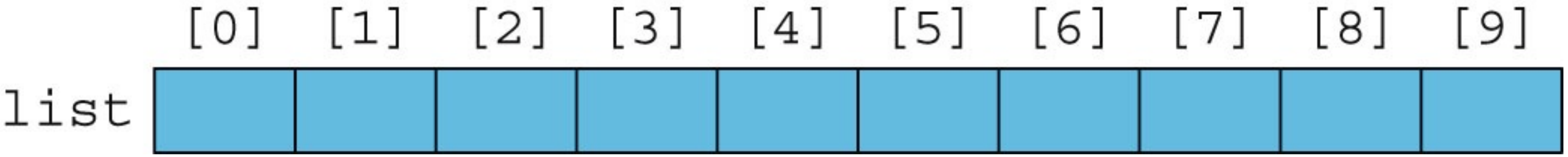


```
arrayName[indexExp]
```

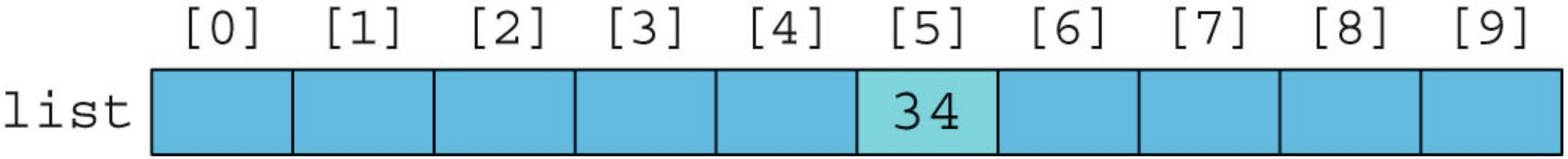
- `indexExp`: called the **index**
  - An expression with a nonnegative integer value
- Value of the index is the position of the item in the array
- **[ ]** : array subscripting operator
  - Array index always starts at 0

Accessing Array Components (cont'd.)

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

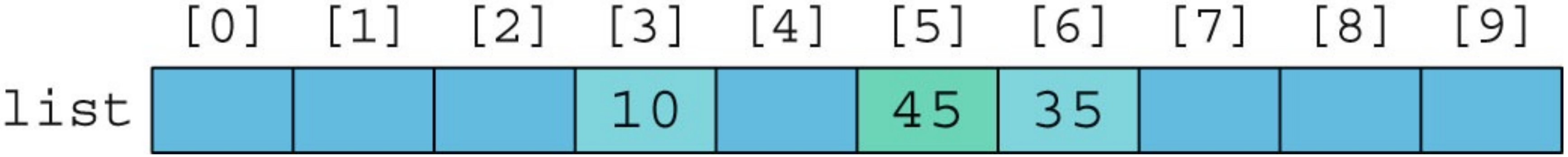


```
list[5] = 34;
```



Accessing Array Components (cont’d.)

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



	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
list				10		45	35			

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**

(see `arrays.c` in *Code Samples and Demonstrations* in *Canvas*)

Arrays

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

☠ *common source of bugs* ☠
referencing first element as a[1]

(see `arrays.c` *and* `commute.c` in *Code Samples and Demonstrations* in *Canvas*)

COLLEGE OF COMPUTING AND INFORMATICS

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];
```

(see `array_initialization.c` and `array_init_warn.c` *Code Samples and Demonstrations* in *Canvas*)

- 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)

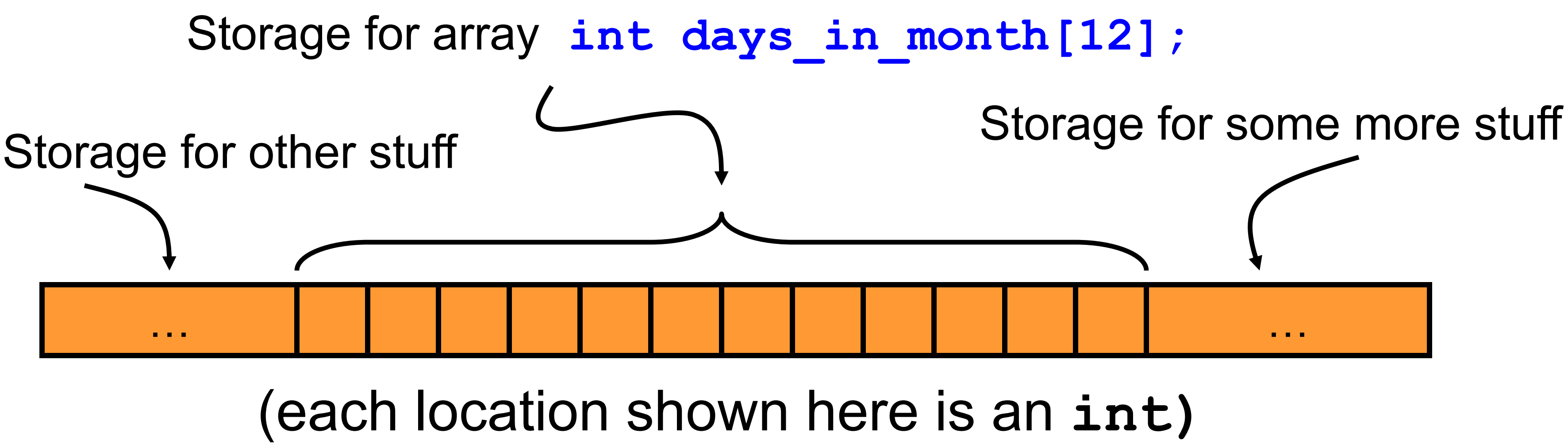
- **Implicit** length + **full** initialization:

```
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.

(see `array_initialization.c` and `array_init_warn.c` *Code Samples and Demonstrations* in *Canvas*)

Memory Layout and Bounds Checking


Storage for array `int days_in_month[12];`
Storage for other stuff
Storage for some more stuff
…
…
(each location shown here is an `int`)

- 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

(see **`array_bounds.c`** in *Code Samples and Demonstrations* in *Canvas*)

☠ *common source of bugs* ☠

referencing outside the declared bounds of an array

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??

(see `var_array.c` in *Code Samples and Demonstrations* in *Canvas*)

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];
```

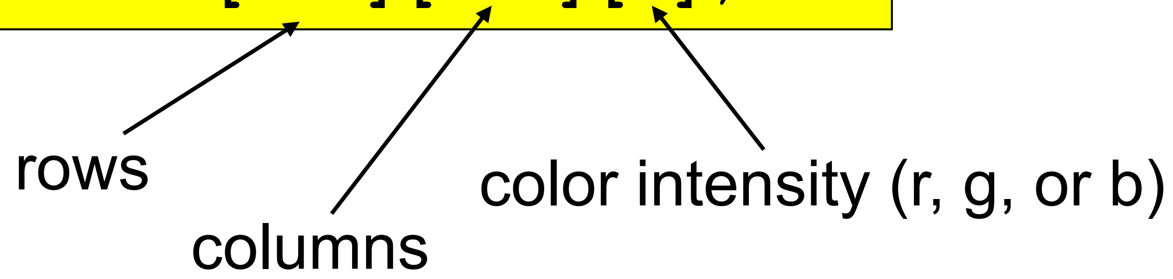
Will not work! Compile error!

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

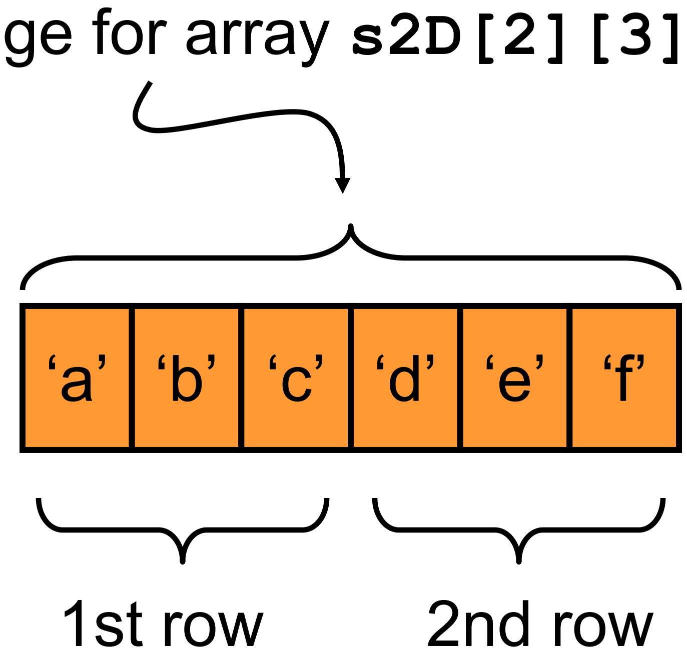
```
int i[][3] =
{ {0, 1, 2}, {4, 5, 6} };
```

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?

Functions and Arrays

Arrays as Function Arguments

- An array can be passed as an **input argument**
- You can specify the array length **explicitly** in the **function declaration**
- Example:

```
void getdays (int months[12])
{
    …
}
```

```
void getdays (int years[10][12])
{
    …
}
```

Arrays as Arguments (cont’d)

- Make **sure** actual argument lengths **agree** with formal argument lengths!
  - – will generate compiler errors otherwise
- Example:

```
int years[5][12];
…
result = getdays (years);
```

why not `years[5][12]` here?

(see `functions_2.c` in *Code Samples and Demonstrations* in *Canvas*)

Omitting Array Sizes

- **Implicit** length for the **first dimension** of a formal parameter is **allowed**
- However, you **cannot omit** the length of **other dimensions**

OK

```
void days (int years[][12])
{
    …
}
```

NOT OK

```
void days (int years[10][])
{
    …
}
```

Dynamic Array Size Declaration

- Q: How can you tell how big the array is if its size is implicit?
- A: You provide array size as an **input parameter** to the function
- Example:

```
void days (int nm, int months[])
{ … }
```

OR

```
void days (int nm, int months[nm])
{ … }
```

Make sure the size parameter comes **before** the array parameter.

Dynamic Array Size… (cont’d)

```
void days(int ny, int nm, int years[ny][nm])
{   …
    for ( i = 0 ; i < ny ; i++)
        for ( j = 0; j < nm ; j++)
            dcnt += years[i][j];
    …
}
```

Make sure sizes are consistent with array declaration

problem here!

```
int years[10][12];
…
(void) days(20,12, years);
```

☠ *common source of bugs* ☠

mismatches in array size declarations

Arrays as Parameters

- **Arrays are passed BY REFERENCE**, not by value
  - i.e., the **callee** function **can modify** the caller’s array values
- Therefore, if you update values in an array passed to a function, you **are** updating the caller’s array

```
int years[10][12];
…
(void) changedays(years);
…
void changedays (int inyears[10][12])
{ …  inyears[1][7] = 29;   … }
```

☠ *common source of bugs* ☠
confusion about call by reference vs. call by value

(see `array_params.c` in *Code Samples and Demonstrations* in *Canvas*)

COLLEGE OF COMPUTING AND INFORMATICS

Arrays Cannot be Return Values

- **Functions *cannot* return *arrays***, nor can they return **functions**
  - (although they can return **pointers** to both)

```
int main(void) {
    char s[100];
    …
    s[] = readstring();
    …
}

char readstring() [100]  {
     …
}
```

Not legal – do not try!

(see `array_return.c` in *Code Samples and Demonstrations* in *Canvas*)

Character Strings

Character Strings

- **Strings** (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:

☠ *common source of bugs* ☠

failure to null terminate a string

```
char s1[] = “hope”;
char s2[] = { ‘h’, ‘o’, ‘p’, ‘e’ };
```

In the first style, the string is implicitly null-terminated by the compiler. The array is 5 characters long because the null character (`’\0’`) is added to mark the end of the string.

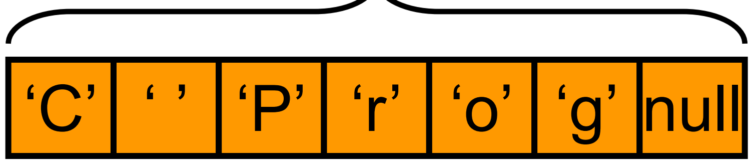
(see **`string.c`** in *Code Samples and Demonstrations* in *Canvas*)

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);
```

Storage for array **s3[]**



‘C’	‘ ’	‘P’	‘r’	‘o’	‘g’	null

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.” ;
```

Array Operators

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); /* ? */
```

what are these sizes?

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

References

- S. J. Matthews, T. Newhall and K. C. Webb, *Dive into Systems*, Version 1.2. Free online textbook, available at: https://diveintosystems.org/book/
- 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.