Arrays in C

C Programming and Software Tools
N.C. State Department of Computer Science
Contents

• Declaration
• Memory and Bounds
• Operations
• Variable Length Arrays
• Multidimensional Arrays
• Character Strings
• `sizeof` Operator
Arrays

• Almost any interesting program uses for loops and arrays

• $a[i]$ refers to $i^{th}$ element of array $a$
  – numbering starts at 0

• Specification of array and index is commutative, i.e., $a[i]$ references the same value as $i[a]$!

```
days_in_month[0] = 31;
days_in_month[1] = 28;
```
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.
Declaring 1-D Arrays

- Explicit length, nothing initialized:

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

- Explicit length, **fully** initialized:

  ```
  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??
Declaring 1-D... (cont’d)

• Implicit length + full initialization:

```c
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
Declaring 1-D... (cont’d)

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

- Two styles:

```c
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};

```
Declaring 1-D... (cont’d)

- Implicit array length and partial initialization?

```c
char first_initial[] = {
};
```

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 \texttt{days\_in\_month[216]} or \texttt{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, e.g.:

```c
int exam1_grade[NUM_STUDENTS],
    hw1[NUM_STUDENTS],
    hw2[NUM_STUDENTS],
    hw_total[NUM_STUDENTS];

for (int j = 0; j < NUM_STUDENTS; j++) {
    exam1_grade[j] = 100;
    hw_total[j] = hw1[j] + hw2[j];
}
```
Variable Length Arrays

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

```c
int i, szar;

printf("Enter # of months in year: ");
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

```c
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];
```

Won’t work! Compile error!
Multi-Dimensional ("M-D") Arrays

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

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

• Referring to one element of a multi-dimensional array:

```c
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

```c
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.

```c
/* 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

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

OK

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

NOT OK
Memory Layout of M-D Arrays

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

Storage for array `s2D[2][3]`

```
| 'a' | 'b' | 'c' | 'd' | 'e' | 'f' |
```

1st row \hspace{1cm} 2nd row

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

```c
char s1[] = "csc230";  // In the first style, the string is implicitly null-terminated by the compiler  
char s1[] = { 'c', 's', 'c', '2', '3', '0' };  
```

In the first style, the string is **implicitly null-terminated** by the compiler

– i.e., the array is **7** 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 simply look for a null character to recognize the end of the string
  - Ex.: `printf()` prints string of arbitrary length using format specifier `%s` (string must be null terminated)

```c
char s1[] = "csc203";
printf ("I’m in %s\n", s1);
```
Character String Concatenation

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

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

• Output of execution is:

```
Now is the time
```

```c
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`

```c
#include <stdlib.h>
size_t tsz1, tsz2, tsz3;
int a;
float b[100];

tsz1 = sizeof (a);
tsz2 = sizeof (b);
tsz3 = sizeof (b[0]);
```

What are these values?
The `sizeof` Operator (cont’d)

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

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

`sizeof()` is evaluated at compile time for statically allocated objects.
Exercise 08a

Reverse 10

• Write a program that reads 10 integers and prints them in reverse order. Use an array of course.

```shell
% ./reverse10
2 3 5 7 11 13 17 19 23 29
29 23 19 17 13 11 7 5 3 2
```
Any Questions?