Arrays, strings, and functions

Goals of this Lecture

- Helps you learn about:
  - Arrays and strings
  - Functions
    - Recursive functions
  - Some pointer concept, but we will defer the details to next lecture
- Subset of what the book covers
  - Important to read the book chapters

The Array Data Type

- Definition
  - Data structure containing a number of data values
  - Data values = *elements*
- Array declaration (one-dimensional array)

```
TYPE Array-name[size];
```

- Examples

```
#define N 20

int a[10];     /* array of 10 integers a[0]…a[9] */
int a[N];      /* array of N integers: a[0]…a[N-1] */
char msg[10]; /* array of 10 chars */
char *msg[N]; /* array of N char pointers */
```

Array Indexing

- The elements of an array of length *n* are indexed from 0 to *n* – 1.

- Expressions of the form `a[i]` are lvalues, so they can be used in the same way as ordinary variables:

```
a[0] = 1;
printf("%d\n", a[5]);
++a[i];
```

- In general, if an array contains elements of type *T*, then each element of the array is treated as if it were a variable of type *T*.

Initialization Examples

- `int a[5] = {1, 2, 3, 4, 5};`
  - {1, 2, 3, 4, 5} is called ***array initializer***
  - `a[0]=1, a[1]=2, a[2]=3, a[3]=4, a[4]=5`
- `int a[5] = {1, 2, 3};`
  - `a[0]=1, a[1]=2, a[2]=3, a[3]=0, a[4]=0`
  - `a[N] = {0};` /* set `a[0]…a[N-1]` to 0 */
  - `a[N] = {};` /* illegal, at least one init value needed */
- `int a[] = {1,2,3,4,5};`
  - `int a[5] = {1,2,3,4,5};`
- Designated initializers (C99)
  - `a[50] = {[2] = 29, [9] = 7, [3] = 3*7 };`
  - Rest of the elements are assigned 0

Type and sizeof

- `int a[5];`
  - What is the type of `a`?
    - The type of a is an integer array
  - What is the type of `a[3]`?
    - The type of a[3] is integer
  - `sizeof(array)` returns # of memory bytes for array
    - `sizeof(a)`, `sizeof(a[3])`

```
#define N 10
#define SIZEOFARRAY(x) (sizeof(x)/sizeof(x[0]))
…

int a[N];
for (i = 0; i < SIZEOFARRAY(a); i++)
      a[i] = 0;
```

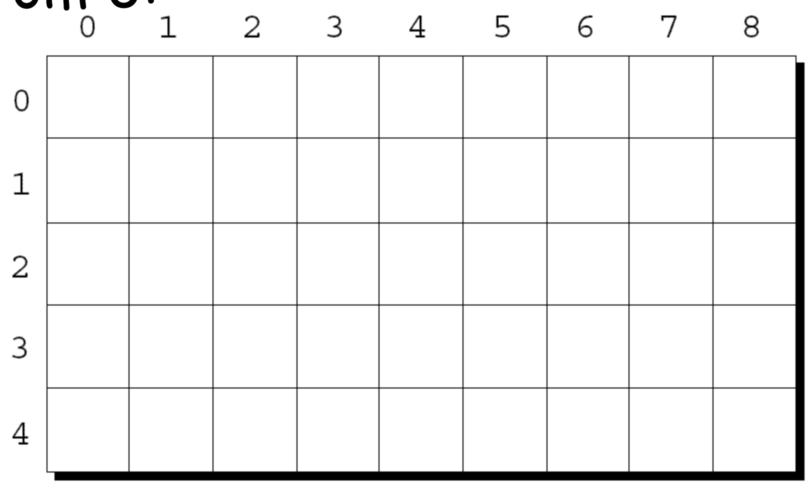
Multidimensional Arrays

- An array may have any number of dimensions.

- The following declaration creates a two-dimensional array (a *matrix,* in mathematical terminology):

```
int m[5][9];
```

  – `m` has 5 rows and 9 columns. Both rows and columns are indexed from 0:

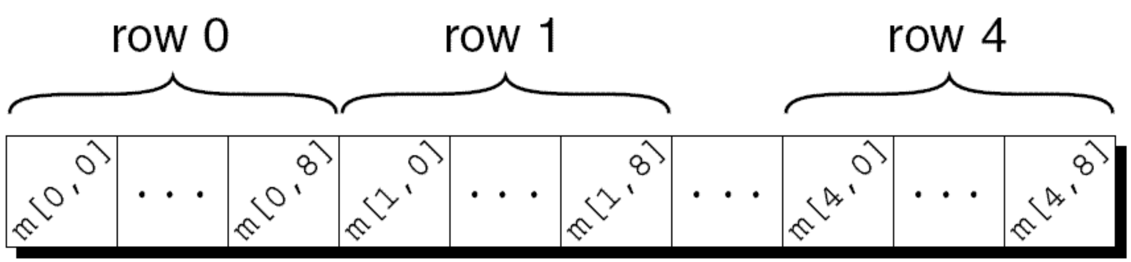


Multidimensional Arrays

- To access the element of `m` in row `i`, column `j`, we must write `m[i][j]`.
- The expression `m[i]` designates row `i` of `m`, and `m[i][j]` then selects element `j` in this row.
- Resist the temptation to write `m[i,j]` instead of `m[i][j]`.
- C treats the comma as an operator in this context, so `m[i,j]` is the same as `m[j]`.

Multidimensional Arrays

- Although we visualize two-dimensional arrays as tables, that's not the way they're actually stored in computer memory.
- C stores arrays in ***row-major order,*** with row 0 first, then row 1, and so forth.
- How the `m` array is stored:



Initializing a Multidimensional Array

- `int a[2][5]={{1,2,3},{6,7,8,9,10}};`
  - `a[0][0]=1, a[0][3]=0, a[0][4]=0, a[1][3]=9`
- C99 designated initializers
  - `int a[2][5] = {[0][0] = 1, [1][1] = 1};`
- C99 variable-length arrays

```
int n;
…
scanf("%d", &n);
…
int a[n]; /* size of array depends on n */
```

Constant Arrays

- An array can be made “constant” by starting its declaration with the word `const`:

```
const char hex_chars[] =
  {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
   'A', 'B', 'C', 'D', 'E', 'F'};
```

- An array that’s been declared `const` should not be modified by the program.

```
hex_chars[0] = ‘k’; /* compile error*/
```

Constant Arrays

- Advantages of declaring an array to be `const`:
  - Documents that the program won't change the array.
  - Helps the compiler catch errors.

- `const` isn't limited to arrays, but it's particularly useful in array declarations.
  - Example: ready-only table (`log[x]`, for integer `x`)

Character Array

- `char x[4] = {‘a’, ‘b’, ‘c’, ‘\0’};`
  - `x[0]=‘a’, x[1]=‘b’, x[2]=‘c’, x[3]=‘\0’`
  - `char x[4] = {‘a’, ‘b’, ‘c’};`
    - `x[3]=0 or x[3]=‘\0’`
  - `char x[] = {‘a’, ‘b’, ‘c’, ‘\0’};`
    - `[]`: compiler determines the size
  - `char x[4] = “abc”;`
    - `“abc”` is **not a string literal** when used as init value for a char array. `“abc”` is abbreviation for `{‘a’,‘b’,’c’,‘\0’}`.
  - `char x[] = “abc”;` **/* same as** `char x[4]=“abc”;` ***/**

String Literals

- A ***string literal*** is a sequence of characters enclosed within double quotes:

  `"When you come to a fork in the road, take it.“`

- String literals may contain escape sequences.

- For example, each `\n` character in the string

  `"Candy\nIs dandy\nBut liquor\nIs quicker.\n  --Ogden Nash\n"`

  causes the cursor to advance to the next line:

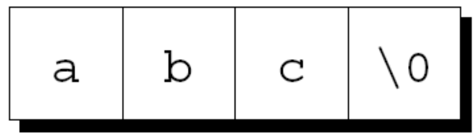
```
Candy
Is dandy
But liquor
Is quicker.
  --Ogden Nash
```

How String Literals are Stored

- When a C compiler encounters a string literal of length *n* in a program, it sets aside *n* + 1 bytes of memory for the string.
  - This memory will contain the characters in the string, plus one extra character—the ***null character***—to mark the end of the string.
  - The null character is a byte whose bits are all zero, so it's represented by the `\0` escape sequence.

How String Literals are Stored

- The string literal `"abc"` is stored as an array of four characters:



- The string `""` is stored as a single null character:

- What about `“abc\0”`?
  - `sizeof(“abc\0”)`?
  - `strlen(“abc\0”)`?

Operations on String Literals

- We can use a string literal wherever C allows a `char *` pointer:

```
char *p;
p = "abc";
```

- This assignment makes `p` point to the first character of the string.
  - “abc” evaluates to the address of the first character of the string

Operations on String Literals

- String literals can be subscripted:

```
char ch;
ch = "abc"[1];
```

The new value of `ch` will be the letter `b`.

```
char *p = “abc”;
ch = p[1]; /* ch = *(p+1); */
```

- A function that converts a number between 0 and 15 into the equivalent hex digit:

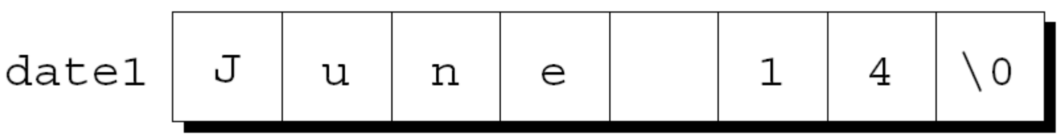
```
char digit_to_hex_char(int digit)
{
  return "0123456789ABCDEF"[digit];
}
```

Initializing a String Variable

- A string variable can be initialized at the same time it's declared:

```
char date1[8] = "June 14";
```

- The compiler will automatically add a null character so that `date1` can be used as a string:



- `"June 14"` is not a string literal in this context.
- Instead, C views it as an abbreviation for an array initializer. (slide 13)

Initializing a String Variable

- If the initializer is too short to fill the string variable, the compiler adds extra null characters:

```
char date2[9] = "June 14";
```

Appearance of `date2`:

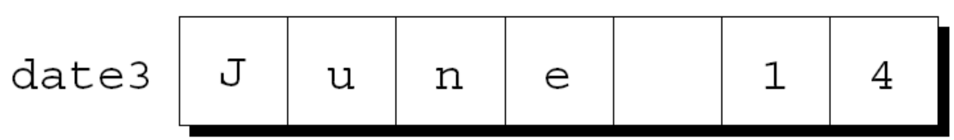
date2	J	u	n	e		1	4	\0	\0

Initializing a String Variable

- An initializer for a string variable can't be longer than the variable, but it can be the same length:

```
char date3[7] = "June 14";
```

- There's no room for the null character, so the compiler makes no attempt to store one:



Initializing a String Variable

- The declaration of a string variable may omit its length, in which case the compiler computes it:

```
char date4[] = "June 14";
```

- The compiler sets aside eight characters for `date4`, enough to store the characters in `"June 14"` plus a null character.

- Omitting the length of a string variable is especially useful if the initializer is long, since computing the length by hand is error-prone.

Character Arrays versus Character Pointers

- The declaration `char date[] = "June 14";` declares `date` to be an *array,*
- The similar-looking `char *date = "June 14";` declares `date` to be a *pointer.*
- Thanks to the close relationship between arrays and pointers, either version can be used as a string.

Character Arrays versus Character Pointers

- However, there are significant differences between the two versions of `date`.
  - In the array version, the characters stored in `date` can be modified. In the pointer version, `date` points to a string literal that shouldn't be modified.
  - In the array version, `date` is an array name. In the pointer version, `date` is a variable that can point to other strings.

Character Arrays versus Character Pointers

- The declaration `char *p;` does not allocate space for a string.
- Before we can use `p` as a string, it must point to an array of characters.
- One possibility is to make `p` point to a string variable:

```
char str[STR_LEN+1], *p;
p = str;
```

- Another possibility is to make `p` point to a dynamically allocated string.

Functions

- Function: a series of statements that have been grouped together and given a name.
  - Each function is a small program
  - Building blocks of larger C program
- Function definition

return-type function-name (*parameters*)
```
{
  declarations
  statements
}
```

  - Function may not return arrays, but can return others.
  - `void` return type indicates it does not return a value.
  - If the return type is omitted in C89, the function is assumed to return a value of type `int`.
  - In C99, omitting the return type is illegal.

Examples

- Calculating the average of two double values

```
double average(double a, double b)
{
  return (a + b) / 2;
}
```

- See if n is a prime number

```
int is_prime(int n)
{
  int divisor;
  if (n <= 1) return FALSE;
  for (divisor = 2; divisor * divisor <= n; divisor++)
       if (n % divisor == 0)
          return FALSE;
  return TRUE;
}
```

Function Calls

- Function name followed by a list of arguments in parentheses

```
double average(double a, double b)
{
  return (a + b) / 2;
}

…
double avg = average(x, y);
```

- What happens under the hood?
  - Before executing the function body, parameters are assigned with the passed arguments
  - a = x; b = y; /* executed before executing other statements */

Function declarations

- Before function call, the compiler needs to know the type of the function

```
return-type function-name (params);
```

```
double average(double a, double b); /* declaration */

int main(void)
{
  double x, y;
  scanf("%lf %lf", &x, &y);
  printf("Average of %g and %g: %g\n", x, y, average(x,y));
  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```

Recursive Function

- Function that calls itself in its body
- Example: factorial of n (or n!)

```
int fact(int n)
{
  if (n <= 1)
     return 1;
  return n * fact(n-1);
}
```

Not correct for large or negative n

- fact(3);
  - return `3 * fact(2)`
  - return `3 * (2 * fact(1))`
  - return `3 * (2 * 1)`

Recursive Function

- Useful in *divide-and-conquer*
  - Divide the work into smaller pieces
  - Smaller pieces are handled with the same algorithm
- Examples
  - factorial of n: `fact(n) = n * fact(n-1)`
    - `fact(n-1)` is solved in the same way
  - Quicksort of n values
    - Pick e among n values
    - Partition the values into two groups, A and B
    - All values in A are less than or equal to e
    - All values in B are larger than or equal to e
    - Run Quicksort for A and Quicksort for B

Summary

- Array: a collection of elements
  - Initialization, sizeof(), multi-dimensional
  - const array, char array
- Function
  - Building block of a program
  - Declaration needed before function call
  - Recursive function: calls itself in the body