Arrays and Strings

Goals of this Lecture

- Help you learn about: Arrays and strings
- Arrays: a group of elements of the same type
  - Type, and element access
  - Initialization, and multidimensional arrays
- Strings: character array + null char
  - No special operators for strings in C
  - Operations through C runtime library functions
- Will use some pointer concept, but we will defer the detail to next lecture

The Array Data Type

- Data structure containing a number of data values
  - Data values = *elements*
  - Elements are of the same type
- 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

- Array of size n: indexed from 0 to n-1
- `a[i]` accesses i-th element (lvalue)
  - lvalue: storage location that potentially allows assignment
    - All variables (including const) are lvalues
    - Constants (e.g., 34) are not lvalue (called rvalue)

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

  - An element in an array of type T is treated as type T

```
int a[10];     // a: integer array type
j = a[3] + 1;  // a[3]: integer type
```

Array Initialization

```
int a[5] = {1, 2, 3, 4, 5};
//  {1, 2, 3, 4, 5} is called  array initializer
// same as 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 initialization value needed */
```

```
int a[] = {1,2,3,4,5};
// compiler counts # of initializers, and fills in
// the size when [] is used.
// It's the same as int a[5] = {1,2,3,4,5};
```

- Designated initializers (C99)

```
int a[50] = {[2] = 29, [9] = 7, [3] = 3*7 };
// rest of the elements are assigned 0
```

Type and sizeof

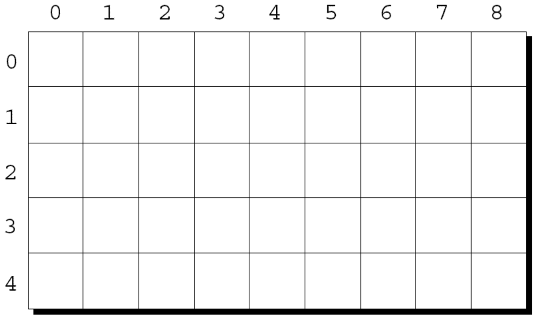
```
char a[5];
```

- What is the type of **`a`**?
  - Answer: it's a **char array** type
- What is the type of **`a[3]`**?
  - Answer: it's a **char** type
- **`sizeof(array)`** returns # of memory bytes for array
  - **`sizeof(a)?, sizeof(a[3])?`**

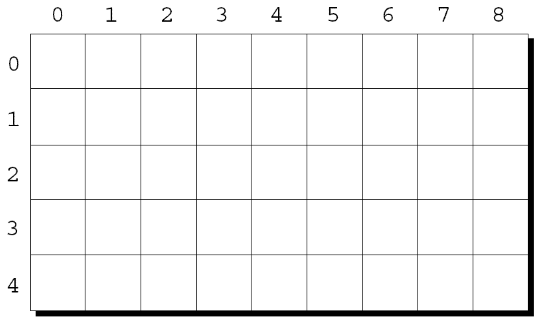
```
#define N 10
#define SIZEOFARRAY(x) (sizeof(x)/sizeof(x[0]))
…

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

Multidimensional Arrays

- Can have an arbitrary number of dimensions
- Two-dimensional array (or a *matrix*)
  - `int m[5][9];`
    - 5 rows and 9 columns
    - Rows and columns are indexed from 0
  - `m[i][j]`: element of m in row i, column j
    - `m[i]`designates row i of m
    - `m[i][j]` selects j element in this row
  - `m[i][j]!=m[i,j]`
    - Since `m[i, j] = m[j]`
    - Comma operator: sequential execution

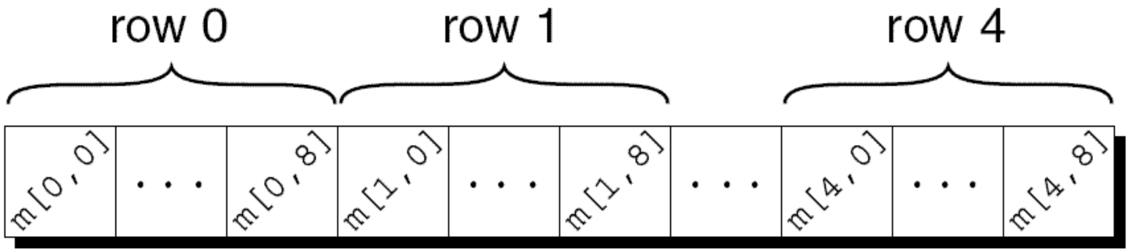


Memory Representation of Multidimensional Arrays



- `int m[5][9];`
- Table is a conceptual model
- Not how they are stored in memory

- C stores arrays in ***row-major order***
  - Row 0 first, row 1, and so forth
- How the array `m` 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
  - Allows initialization of selected elements
  - Other elements are initialized to 0

```
int a[2][5] = {[0][0] = 1, [1][1] = 1};
```

- C99 variable-length arrays
  - Array size can be dynamic for C99

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

Constant Arrays

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

- []: size is automatically filled in by compiler (=16)
- `const` doesn’t allow changing any value (read-only)

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

- Why do we use `const`?
  - Tell that the program will not change the array
  - Helps the compiler catch ***errors***
  - Use `const` as much as possible where appropriate!
- `const` isn’t limited to arrays
  - But particularly useful in array declarations
  - e.g., read-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’ ( 0 == ‘\0’)

char x[] = {‘a’, ‘b’, ‘c’, ‘\0’}; // size: 4
char x[] = {‘a’, ‘b’, ‘c’};       // size: 3

char x[4] = “abc”;
// “abc” is not a string literal when used as
// initialization value for a char array.
// “abc” is just abbreviation for {‘a’,’b’,’c’,‘\0’}.

char x[] = “abc”; /* char x[4] = “abc”; */
```

String Literal

- A sequence of chars enclosed within double quotes
  - e.g., **“hello world”**
  - Also called a string constant
- May have escape sequences

```
printf("Candy\nIs dandy\nBut liquor\nIs quicker.\n  --Ogden Nash\n”);
```

```
$./a.out
  Candy
  Is dandy
  But liquor
  Is quicker.
    --Ogden Nash
```

How String Literals are Stored

- C compiler meets an n-char string literal in a program
  - It sets aside *n* + 1 bytes of memory for the string
  - n bytes for characters + 1 extra character (**null char**)
  - The null character marks the end of a string (`'\0'` or 0)
- `"abc"` takes up 4 bytes in memory

a	b	c	\0

- "" is a single null character

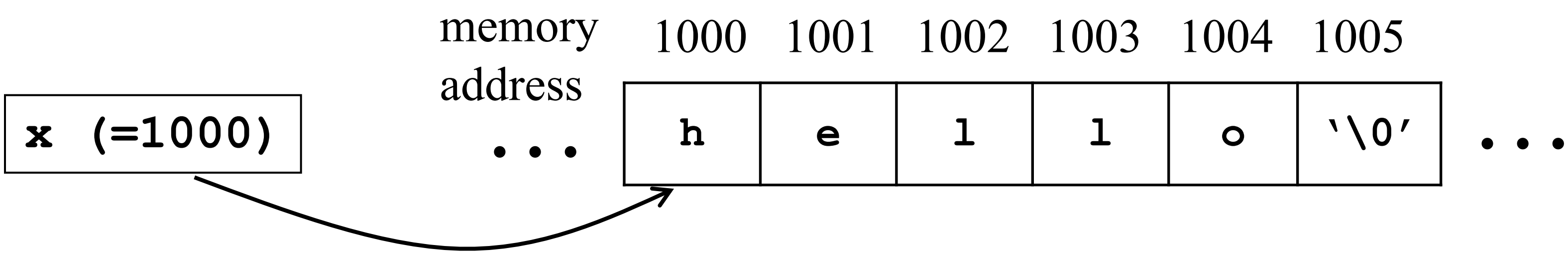
\0

- What about "`abc\0`"?
  - **`sizeof`**`("abc\0")`?
  - **`strlen`**`("abc\0")`?

strlen() returns the # of chars of the string (argument) excluding the null character

String Literal and Char Pointer

```
char *x = "hello";
```



x (=1000)

memory address

...	1000	1001	1002	1003	1004	1005	...
	h	e	l	l	o	'\0'	

- A string literal evaluates to the memory address of the first character
    - A string literal can be assigned to a character pointer variable
    - A char pointer variable (x) holds an address of a character
    - A pointer can be subscripted (very similar to an array)
    - e.g., x[1] accesses 'e' and x[4] accesses 'o'
- More details on C pointers will be discussed in the next lecture

Function Parameter Passing

- strlen() is a C standard library function
  - #include <string.h>
- One implementation of strlen()

```
int strlen(char *x)
{
  int n = 0;
  while (x[n] != ‘\0’)
    n++;
  return n;
}
```

```
int k = strlen(“hello”);
```

- x = “hello”; is executed before function body
  - x holds the address of the first char of “hello”

Operations on String Literals

- String literals can also be subscripted:

```
char ch;

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

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

```
 char *p = “abc”;

 ch = p[1];
```

- A function that converts a number (0~15) into the equivalent hex digit:

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

Initializing a String Variable

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

date1	J	u	n	e		1	4	\0

- `"June 14"` is **not a string literal** in this context
  - An abbreviation for an array initializer
  - {‘J’, ‘u’, ‘n’, ‘e’, ‘ ‘, ‘1’, ‘4’, ‘\0’};

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

Too big: compiler fills in 0

date2	J	u	n	e		1	4	\0	\0

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

Too small to squeeze in null char

date3	J	u	n	e		1	4

Initializing a String Variable

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

- Compiler determines the length of date4 if the size is unspecified (8 bytes including the null)
- Useful if the initializer is long
  - Manual counting of # of chars is error-prone

```
  char date5[8] = “June 14”;
  char date6[8] = {‘J’, ‘u’, ‘n’, ‘e’, ‘ ’, ‘1’, ‘4’};
  char date7[]  = {‘J’, ‘u’, ‘n’, ‘e’, ‘ ’, ‘1’, ‘4’};
```

- Difference among **`date4`**, **`date5`**, **`date6`**, **`date7`**?
  - sizeof(date4)? sizeof(date5)? sizeof(date6)? sizeof(date7)?

Char Array vs. Char Pointer

```
   char date1[8] = "June 14";
   char *date2 = "June 14";
```

- **`date1`** is an array while **`date2`** is a character pointer
- Similarity between them
  - Reading characters: **`date1`**[k], **`date2`**[k]
- Difference between them
  - Array (date1) can modify characters vs. date2 can’t
    - Array elements are allocated at ***read-write*** memory section
    - String literals are allocated at ***read-only*** memory section
  - Array name (date1) can’t change its own value
  - Char pointer (date2) is a variable, and it can change its own value

```
   date1[3] = ‘K’;      // valid
   date2[3] = ‘K’;      // invalid, runtime error
   date2++; date2 += 2; // valid
   date1++; date1 += 2; // invalid, compile error
   date2 = date1;       // valid, date1 evaluates to addr of first char
   date2[3] = ‘K’       // valid, date2 points to modifiable area
```

Summary

- Array: a collection of elements of the same type
  - Array initialization, sizeof()
  - Row major layout of multi-dimensional arrays
  - Constant array
- C string: an array of chars terminated by null
  - No special type for a string
  - Warning: an array of chars that don’t end with null
- Character array vs. character pointer
  - Arrays and pointers are closely related
  - More details on pointers will be discussed in the next lecture