Arrays and Pointers (Part 1)

Devin J. Pohly <djpohly@cse.psu.edu>

Makefiles: filling the gaps

- When you run make, it *automatically* reads the file named "Makefile"
- Easiest way to set it up is to ask:
  - What am I starting with? (e.g., source code)
  - What commands do I run to get from there to the final result? (e.g., gcc, ld)
  - What does each command give me? (e.g., object files, executable file)
- The rest is just syntax

Makefile Q&A

```
# Specify compiler and settings
CC = gcc
CFLAGS = -Werror -Wall -Wextra

# Set up a phony target to build everything
.PHONY: all

all: addit hello

# Add an additional dependency without affecting
# the recipe
addit: addit.h

# Compile any simple program
%: %.c
	$(CC) $(CFLAGS) -o $@ $^
```

One more thing

- Make already knows how to build certain things
  - Called “implicit rules”
  - Can be left out of a Makefile
  - Example: `%: %.c`
  - BUT for this assignment you need to write out all of your rules
- PROTIP: run `make -p` in a directory with no Makefile for the (big) list.

Aside: the `sizeof` operator

- Usage:

```
int b = sizeof(x);
```

- Gets the size (in bytes) of a variable, based on its type.
  - One way to answer the question "how big is an `int` on my system?"
  - Can be applied to any data type, with a caveat for arrays
  - Will be useful when working with memory

Creating arrays

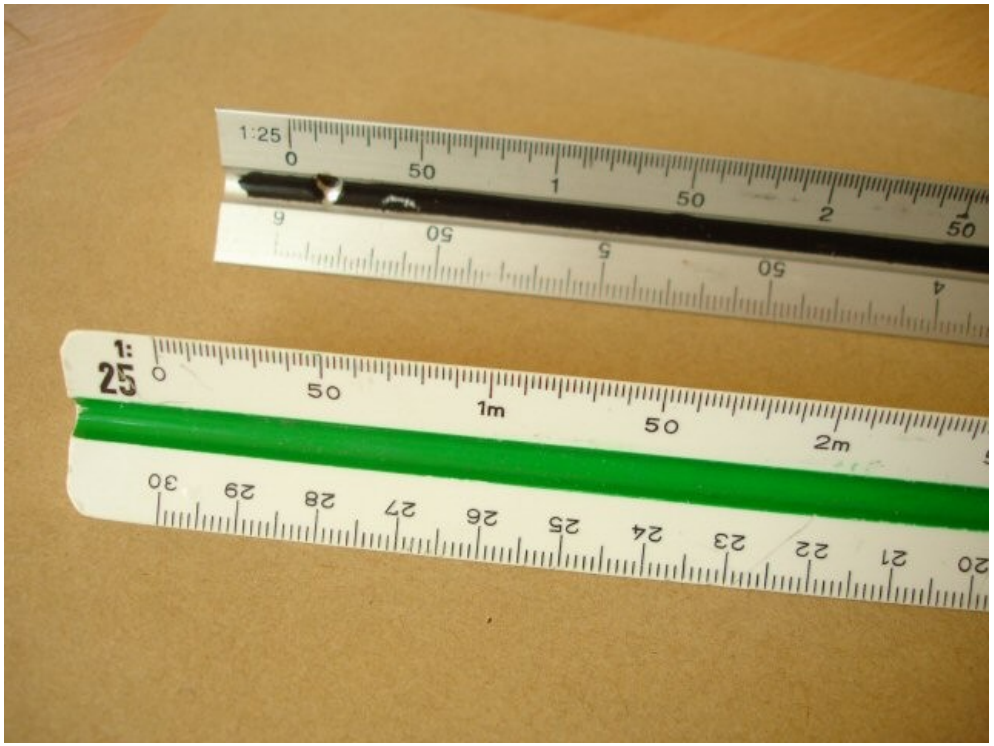
- Declaration:

```
int16_t arr[100];
```

- Allocates a block of raw memory
  - In this example, 200 bytes
  - Inside a function: allocated on the stack
    - Local to the function!
    - Contains garbage until you initialize it
  - Static or global: allocated in the data segment

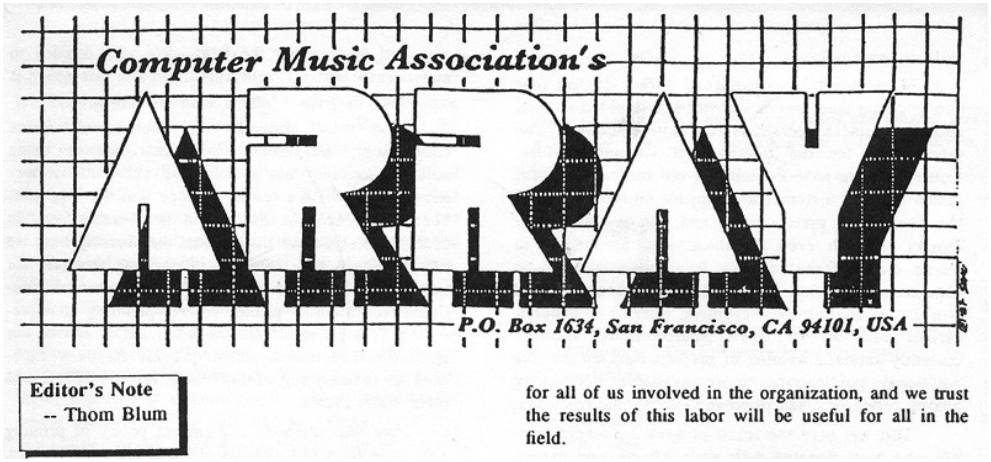
Array length

- Remember: array length is not known at runtime!
  - Sometimes not even at compile time
  - This means sizeof(arr) may not always do what you think!
  - Hint: arrays are pointers
- Keep track of the length
  - Better alternative: `len*sizeof(arr[0])`



Array types

- All of the elements are of one type
  - Ex.: `float nums[32];`
  - Each element is a float
- The array itself is of a different type
  - Part of why sizeof is weird with arrays
  - The type of "nums" is `float[]` (or `float *`)
  - Hint: arrays are pointers



Variable-length arrays

- This is legal and works:

```
int n = 100;
int scores[n];
```

- VLAs introduced in C99
  - But then made optional in C11
  - Not frequently used

Initializing and using arrays

- Can initialize arrays when they are declared
  - Both allocates and fills array
  - Size can be omitted if you use an initializer
  - Padded with 0 or NULL only if you give both a size and an initializer
- Use `arr[i]` to read or write an array element
  - Just like Java or C++, but with no bounds checking

```
int primes[6] = {2, 3, 5, 6, 11, 13};
primes[3] = 7;
primes[100] = 0;  // Legal but BAD!

int fib[] = {1, 1, 2, 3, 5, 8, 13, 21, 34};

int allZeroes[1000] = {0};  // This works, but...
// this initializes to 1,0,0,0,0,...!
int allOnesNOPE[1000] = {1};
```

Initializing elements

- Designated initializers added in C99
  - Allow you to specify which element gets a value
  - All others initialized to 0 or NULL
- GCC extension for range initializers
  - Non-standard, but the cleanest way to initialize many values

```
// Designated initializers
int isPrime[10] = { [2] = 1, [3] = 1, [5] = 1, [7] = 1 };

// Range initializers - both of these will work with GCC
int allZeroes[1000] = { [0...999] = 0 };
int allOnes[1000] = { [0...999] = 1 };
```

Multi-dimensional arrays

- Still just a block of memory!
  - Need to keep track of each dimension
- To declare and initialize:

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

- Add more dimensions as needed

Arrays as parameters

- Using arrays as function parameters is tricky
  - Arrays are effectively passed by reference, not copied
  - Arrays do not know their own size
  - Hint: arrays are pointers


```
int sumAll(int a[]) {
    int i, sum = 0;
    for (i = 0; i < what?; i++)
        sum += a[i];
    return sum;
}

int main(int argc, char *argv[]) {
    int numbers[5] = {3, 4, 1, 7, 4};
    int sum = sumAll(numbers);
    return 0;
}
```

Arrays as parameters

- Attempt: declare the array size in the function
  - Doesn't actually guarantee the array is that size
  - Example: the code below compiles and runs!
  - So this won't work


```
int sumAll(int a[5]) {
    int i, sum = 0;
    for (i = 0; i < 5; i++)
        sum += a[i];
    return sum;
}

int main(int argc, char *argv[]) {
    int numbers[3] = {3, 4, 1};
    int sum = sumAll(numbers);
    return 0;
}
```

Arrays as parameters

- Solution: pass the size as a parameter
  - This is the idiomatic way to pass arrays to a function
  - Used throughout the C and POSIX standard libraries

```
int sumAll(int a[], int size) {
    int i, sum = 0;

    for (i = 0; i < size; i++)
        sum += a[i];
    return sum;
}

int main(int argc, char *argv[]) {
    int numbers[5] = {3, 4, 1, 7, 4};
    int sum = sumAll(numbers, size);

    printf("The sum is %d.\n", sum);

    return 0;
}
```

Returning an array

- Local variables, including arrays, are stack allocated
  - Disappear when the function returns
  - The return statement doesn't copy the array!
    - Hint: arrays are pointers


```
int[] copyarray(int src[], int size) {
    int i, dst[size];   // OK in C99

    for (i = 0; i < size; i++) {
        dst[i] = src[i];
    }
    return dst;   // BUG!
}
```

“Returning” an array

- Solution is to create the array in the caller
  - The function can then manipulate the existing array
  - This works because arrays are effectively passed by reference
    - Why?

```
void copyarray(int src[], int dst[], int size) {
    int i;

    for (i = 0; i < size; i++) {
        dst[i] = src[i];
    }
}
```