WARNING: a big lab this week!

3. chars, arrays and functions

- Chars: *chario.c*
- Arrays: *backwards.c*
- Functions: *average.c*
- Exchanging data via globals: *twiddle.c*
  - DON'T!!! (well, rarely, anyway)
- Arrays as parameters: *twiddle2.c*
- 2D arrays
- Memory layout

chario.c

```
/* Demonstrate the 'char' data type. */

#include <stdio.h>

int main(void)
{
    char someChar = 0;    // A character
    int c = 0;            // An int with a char in its low byte

    someChar = '*';
    printf("someChar = %c\n", someChar); // Print a single char
    printf("Enter a line of text, terminated by 'Enter'\n");

    // Read and print characters until newline or End-Of-File
    c = getchar();                // Get char (cast to int) or EOF
    while (c != '\n' && c != EOF) {
        printf("Char '%c', decimal %d, octal %o, hexadecimal %x\n", c, c, c, c);
        c = getchar();
    }
}
```

Points to note

- *char* variables hold a single 8-bit character or int
- *char* literals use single quotes (strings use double quotes)
- chars use ASCII table for int ↔ character "conversion"
- *getchar*, *putchar* etc have *int* parameters, with char in low byte
  - Allows for "out-of-band" values like EOF == -1
- whether a *char* variable is a short integer or a character depends on how you use it!
  - When you do arithmetic on it, it's an integer
  - If you *printf* it using %c it's a character
  - If you *printf* it using %d, %o or %x it's an integer

backwards.c

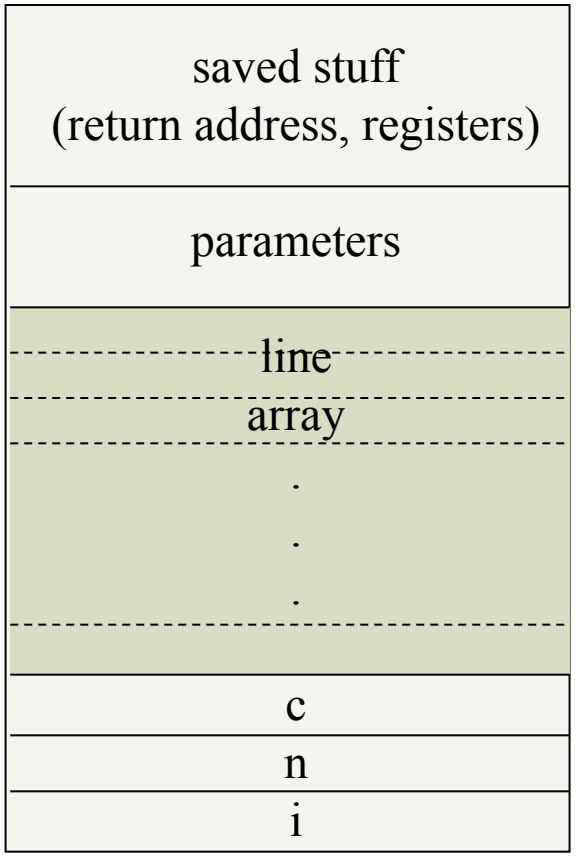
```
#include <stdio.h>
#define N_MAX 100
int main(void)
{
    char line[N_MAX] = {0};   // An array of char.
    int c = 0;        // A generic character, cast to an int
    int i = 0;        // Generic loop index/counter
    int n = 0;        // Number of bytes read
    printf("Variable n requires %zu bytes of memory\n", sizeof n);
    printf("Array line occupies %zu bytes of memory\n", sizeof line);
    printf("Enter a line of text, terminated by 'Enter'\n");
    while ((c = getchar())!= '\n' && c != EOF && n < N_MAX) { // Read a line
        line[i++] = c;    // Add char to line, increment index
                          // [an ok style rule exception]
    }
    n = i;   // Number of bytes read

    for (int i = n - 1; i >= 0; i--) {     // Print it out backwards
        putchar(line[i]);
    }
    putchar('\n');
    // From now on we'll rely on C99 to return 0 (EXIT_SUCCESS) by default
}
```

sizeof gives number of bytes required for a variable or a type

Notes on backwards.c

- Python programmers: arrays are like lists *but*:
  - Can't append or remove items
  - All elements must be of same type
- Array declarations are roughly like Java's, *but*
  - The array *line* is a block of memory on the stack, NOT on the heap
    - No *new* required to allocate space
    - We don't use the heap until week 6!
  - Get zero fill after initialiser runs out
    - But if no initialiser get no initialisation!
      - Except: external arrays *are* initialised to zero
      - See later

Possible Stack Frame for main

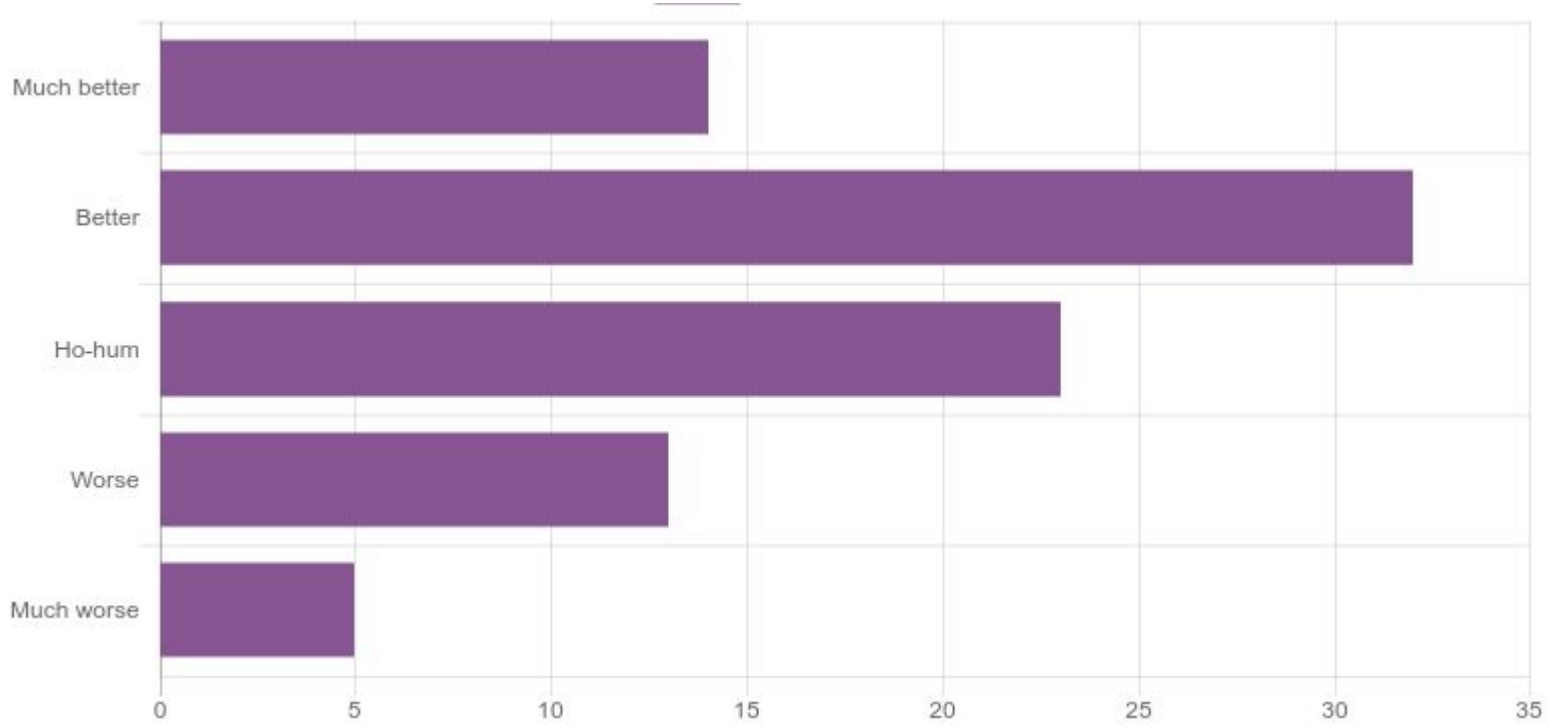


Notes on backwards *(cont'd)*

- Syntax to index array same as in Java but ...
  - No runtime subscript checking
    - If you run off the end of the array anything may happen!
      - e.g., with stack frame picture on prev. slide, you'll read/write the local variables *c*, *n* and *i*.
      - But stack frame layout not predictable. Varies with compiler version and even from run to run (memory-layout randomisation)!
- If you declare an array with an empty size and an initialiser list, the size is the number of initialisers
  - `int data[] = {10, 20, 30}; // A 3-element array`
- *getchar()* gets char from *stdin*. RETURNS AN INT!!
  - Because EOF is an int value (-1), not a char from the file!
- *putchar(c)* outputs a char to *stdout*

Interlude: questionnaire feedback

What did you think of that approach to lecturing?



Specific comments

- It was difficult to focus on the videos knowing they are online anyway and felt tedious. But the question answering is great.
- I would prefer you to just lecture, real time lectures are more engagement.
- I really liked watching the videos in the lecture because it gives us the opportunities to ask questions we have and means we don't have to watch the videos in our own time :)
- The online question forum was great, answered a lot of questions that I wouldn't have actually had until I made a bunch of mistakes and went to google. The video watching format isn't particularly efficient but I think it works as well as lectures
- More of this. Loved it.
- Give us an in lecture problem to solve!
- Working through code examples. Put videos & lecture notes / slides up early, so we can watch beforehand, then ask questions during lecture.
- Spend the lectures going over more complicated and engaging topics. As the basics should be learnt through the videos and labs so this would lead to a more efficient and interesting lecture.
- Cover more content, maybe in extra depth
- Even though it seems pretty redundant going through the content in your videos during the lectures is the best option since most people haven't watched the videos yet and that way we can ask questions.
- As someone who struggles with a short attention span these really helped. Everything was concise, and stopping for questions was a really good idea.

Introducing functions: average.c

```
// Simple demo of use of a function (from King)
#include <stdio.h>

// Declare and define function "average" before using it
double average(double a, double b)  {
    return (a + b) / 2;
}

int main(void)
{
    double x=0, y=0, z=0;  // 3 generic doubles [but generally
                           // this is poor style]
    printf("Enter three numbers: ");
    scanf("%lf%lf%lf", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x, y));
    printf("Average of %g and %g: %g\n", y, z, average(y, z));
    printf("Average of %g and %g: %g\n", x, z, average(x, z));
}
```

OR (average2.c)

```
#include <stdio.h>

double average(double a, double b);  // DECLARATION

int main(void)
{
    double x = 0.0, y = 0.0, z = 0.0;
    printf("Enter three numbers: ");
    scanf("%lf%lf%lf", &x, &y, &z);
    printf("Average of %g and %g: %g\n", x, y, average(x,y));
    printf("Average of %g and %g: %g\n", y, z, average(y,z));
    printf("Average of %g and %g: %g\n", x, z, average(x,z));
}


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

Notes on functions

- Functions similar to Python's or like static methods in Java
  - Declaration requires return type and all parameter types
  - Must declare functions before using them
    - DON'T run with the compiler's guesses at types!
    - DO enable all warnings in the compiler [*gcc –Wall* ...]
      - In C, a warning is usually an error in your program!
        - » At very least it's a style error.
- They're the *only* way to break your code into pieces!
  - But you can (and should) put related functions into separate files and compile them separately (see later)
- Can't nest functions

Array parameters

- Function *average* has scalar parameters, returns a scalar.
- What about arrays as parameters or return values?
- Consider:
  - "Write a function *readName* that reads a line of text up to but not including the first newline."
- In Java or Python we'd return a *string*.
- But C doesn't have (real) strings - only arrays of char.
- How can we return an array of chars from a function?
- Answer: we can't.
  - Or … not easily. But see Dynamic Memory in week 6.
- Instead we must *pre-allocate* space for the "string"

Method 1. Use a global array

Code *twiddle.c*

```
#include <stdio.h>
#include <ctype.h>  // Various character-handling functions defined here

#define MAX_NAME_LENGTH  80
char name[MAX_NAME_LENGTH];  // declares a global (aka "external") variable

// Read a name (or any string) into the "name" array. Terminate it with null.
void readName(void)
{
    int c = 0;
    int i = 0;
    printf("Enter your name: ");
    while ((c = getchar()) != '\n' && c != EOF && i < MAX_NAME_LENGTH - 1) {
        name[i++] = c;
    }
    name[i++] = 0;  // Terminator
}
```

twiddle.c *(cont'd)*

```
// Convert the global "name" string to upper case
void convertNameToUpper(void)
{
    int i = 0;
    while (name[i] != '\0') {
        name[i] = toupper(name[i]);
        i++;
    }
}

// Main program reads name, converts it to upper case and prints it
int main(void)
{
    readName();
    convertNameToUpper();
    printf("Your name in upper case: %s\n", name);
}
```

Notes on twiddle

- Global variables are visible to all functions
  - Lazy! Don't use them!
- Strings are *char* arrays, in which a null character ('\0') denotes end of string.
  - Can be output with `%s` format specifier

Method 2: allocate space in main

Code *twiddle2.c*

```
// Read a name (or any string) into the parameter array.
// `Terminate with null.
void readName(int maxLen, char name [ ])
{
    int c = 0;
    int i = 0;
    printf("Enter your name: ");
    while ((c = getchar()) != '\n' && c != 0 && i < maxLen - 1) {
        name[i++] = c;
    }
    name[i++] = '\0';  // terminator
}

int main(void)
{
    char name[MAX_NAME_LENGTH];
    readName(MAX_NAME_LENGTH, name);
    convertStringToUpper(name);
    printf("Your name in upper case: %s\n", name);
}
```

function *convertStringToUpper* *T.B.S.*

Notes on twiddle2.c

- Arrays are just chunks of memory – when you pass an array as a parameter you pass a *pointer* to that memory (as in Java)
  - Pointers are covered in the next section
  - The whole array is not copied
  - The parameter type doesn't include the array dimension
    - But *do* need dimensions for 2D array
- No way to determine the length of an array so ...
  - Must either pass its length as a parameter too, or
  - Mark the end of the data somehow
    - e.g. the string-terminating null byte.

Interlude

- Check out
  http://qz.com/726338/the-code-that-took-america-to-the-moon-was-just-published-to-github-and-its-like-a-1960s-time-capsule/
- Quiz question:
  - Who was the world's first Software Engineer?
    - Answer is in the above link

2D arrays

- Extension of arrays from 1D to 2D, 3D etc is easy
- Just use more dimensions!

```
char game[3][3] = {{'X', '.', '.'},
                   {'O', 'X', '.'},
                   {'O', 'O', 'X'}};

for (int row = 0; row < 3; row++) {
    for (int col = 0; col < 3; col++) {
        putchar(game[row][col]);
    }
    putchar('\n');
}
```

2D arrays (cont'd)

- 2D arrays are laid out row-wise in memory

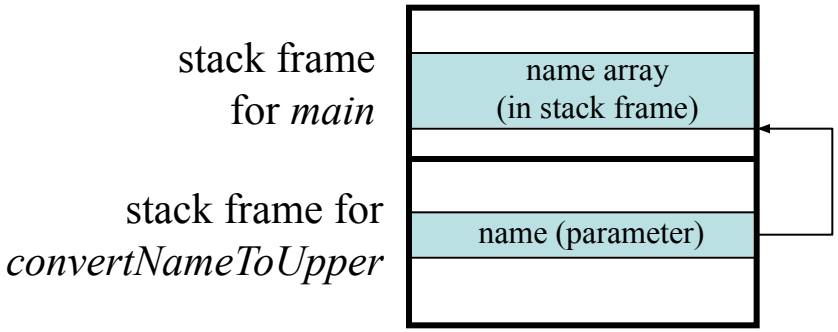
Row0	Row1	Row 2	Row 3	Row 4	…

- 2D indexing is done as:
  - `thing[i][j] => thing[i * NUM_COLUMNS + j]`
- Hence, when passing 2D arrays as parameters, the dimensions are part of the parameter type declaration
  - Or at least the second dimension
- e.g. `void printGame(char game[3][3]) { … }`

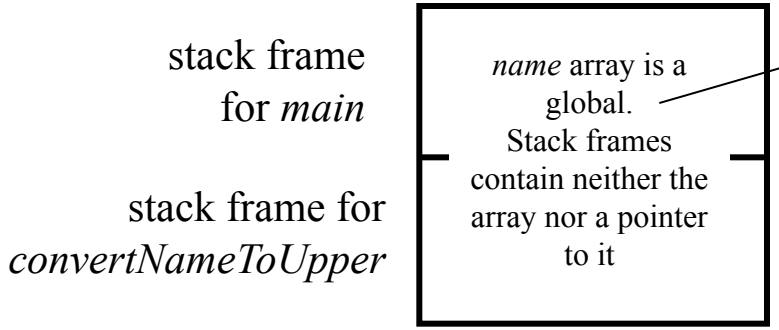
Required

Memory layout (32-bit linux)

- In *twiddle2*:



- In *twiddle:*



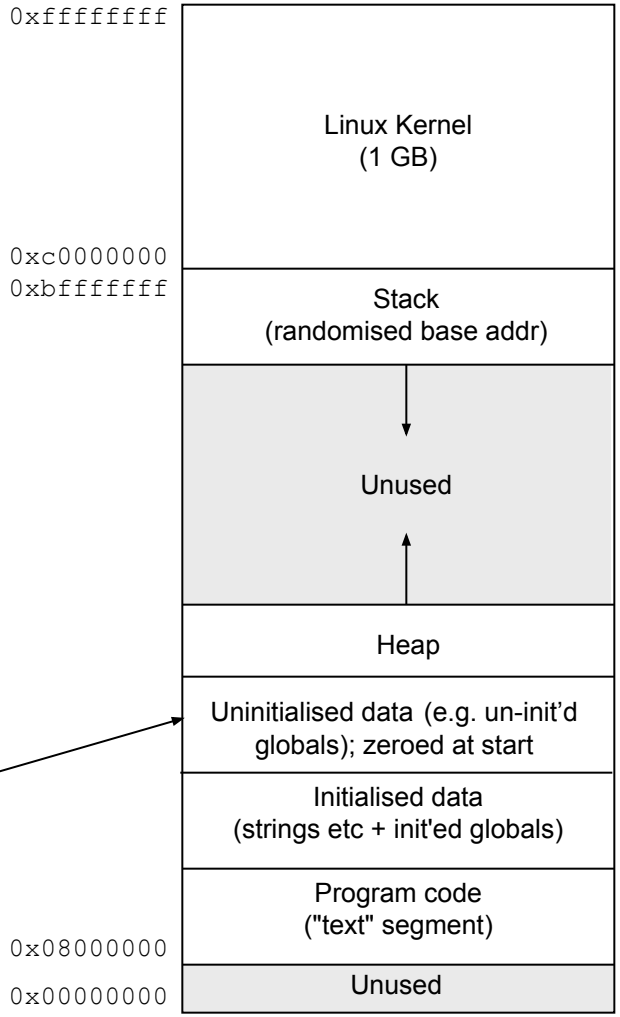


Fig. from *pointers* lab