CS50 Week 2 - Arrays, Strings, and More

Animesh Garg and ChatGPT

1 Introduction

This document contains a comprehensive and **in-depth** set of notes from **Week 2 of CS50**. Building on Week 1's discussion of the C programming language, we dive further into:

- The multi-step **compilation** process
- **Debugging** (via `printf` and the `debug50` debugger)
- The **array** data structure
- How **strings** are stored internally (as arrays of chars plus a NUL terminator)
- **Command-line arguments**, which allow more flexible user input
- **Exit statuses** to indicate success or failure of programs
- Basic **cryptography** concepts, such as the Caesar cipher

Contents

2 Reading Levels

2.1 Motivation

- A real-world problem tackled in Week 2: computing *reading levels*.
- Text can have complexity that corresponds roughly to grade levels.
- We can quantify difficulty by counting words, letters, sentences, etc.

2.2 Sample Readings

- *"One fish, two fish, red fish, blue fish."* → **kindergarten** level
- *"Congratulations! Today is your day!"* (Dr. Seuss) → **3rd grade** level
- A more advanced excerpt (e.g., from *1984*) → **10th grade** level

2.3 Approach in Code

- We'll parse text in a program, using loops and arrays, to measure reading difficulty.
- This sets the stage for Problem Set 2, where you might, e.g., compute an approximate index or measure of readability.

3 Compiling, Step by Step

3.1 Overview

Recall that:

```
make hello
```

automates the translation of source code (C) to machine code. Internally, four steps happen:

1. **Preprocessing**:
   - Lines like `#include <stdio.h>` are preprocessor directives.
   - The compiler effectively copies the contents of those files into your code.
2. **Compiling** (in a narrow sense):
   - Your preprocessed C code is converted to *assembly* instructions.
   - Assembly is a CPU-specific textual language (far lower level than C).
3. **Assembling**:
   - Assembly is turned into raw machine code (bits).
   - Produces an *object file* (often called `a.out` by default).
4. **Linking**:
   - Your object file is combined with object files from libraries.
   - Results in a final executable (e.g. `hello`).

3.2 Example Command

```
clang -o hello hello.c -lcs50
```

is effectively run by `make hello` if you need the CS50 library. `make` hides these details, but it's doing these four steps for you behind the scenes.

4 Debugging

4.1 Rubber Duck Debugging

- Explaining code logic out loud to a rubber duck (or yourself) often helps you find mistakes.
- CS50 provides a *virtual AI Duck* at `https://cs50.ai`.

4.2 `printf` Debugging

- Add `printf` lines to track variable states or loop counters.

  ```
  for (int i = 0; i <= 3; i++)
  {
      printf("i is %i\n", i);
      printf("#\n");
  }
  ```

- Remove these once the bug is diagnosed.

4.3 `debug50`

- In VS Code, set a breakpoint by clicking left of line numbers.
- Then run:

  ```
  debug50 ./myprogram
  ```

- Execution halts at each breakpoint. You can:
  - **Step Over**: run the current line in the same function
  - **Step Into**: dive into a function call
  - **Continue**: run until the next breakpoint or the end
- Inspect local variables (e.g. arrays, loop counters) as your code runs, to confirm logic or see errors in real time.

4.4 Sample Buggy Code (buggy.c)

```
#include <cs50.h>
#include <stdio.h>

// Prototype
void print_column(int height);

int main(void)
{
    int h = get_int("Height: ");
    print_column(h);
}

void print_column(int height)
{
    // BUG: We used <= instead of <
    for (int i = 0; i <= height; i++)
    {
        printf("#\n");
    }
}
```

This might produce one extra `"#"` than intended. Using `debug50` or `printf` debugging clarifies that `i` runs from 0..height inclusive.

5 Arrays

5.1 Concept & Syntax

- An array is a sequence of values in contiguous memory, all the same type.
- Example:

  ```
  int scores[3]; // 3 integers back to back
  scores[0] = 72;
  scores[1] = 73;
  scores[2] = 33;
  ```

5.2 Filling Arrays with Loops

```
const int N = 3;
int scores[N];
for (int i = 0; i < N; i++)
{
    scores[i] = get_int("Score: ");
}
```

5.3 Computing Averages (scores5.c)

```
#include <cs50.h>
#include <stdio.h>

// A function to get the average
float average(int length, int array[]);

int main(void)
{
    const int N = 3;
    int scores[N];

    // Get scores from user
    for (int i = 0; i < N; i++)
    {
        scores[i] = get_int("Score: ");
    }

    // Print the average
    printf("Average: %f\n", average(N, scores));
}

float average(int length, int array[])
{
    int sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += array[i];
    }
    // cast to float to avoid truncation
    return sum / (float) length;
}
```

5.4 Memory Model

- If an `int` is 4 bytes, then `scores[3]` occupies 12 bytes contiguously.

6 Strings

- A *string* in C is effectively an **array of chars** plus a special `'\0'` terminator.

6.1 Basic Example

```
string s = "HI!";
printf("%c\n", s[0]); // 'H'
printf("%c\n", s[1]); // 'I'
printf("%c\n", s[2]); // '!'
printf("%i\n", s[3]); // ASCII 0 => '\0'
```

6.2 NUL Terminator

- The extra '\0' indicates the end of the string.
- So "HI!" is stored as: 'H','I','!', '\0'.

6.3 Multiple Strings

```
string s = "HI!";
string t = "BYE!";

printf("%s\n", s);
printf("%s\n", t);
```

In memory, these might appear consecutively as well.

6.4 Array of Strings

```
string words[2];
words[0] = "HI!";
words[1] = "BYE!";

printf("%s\n", words[0]);
printf("%s\n", words[1]);
```

`words[0]` and `words[1]` are each array-of-chars + terminator.

7 String Length

7.1 Manual Counting

```
int n = 0;
while (s[n] != '\0')
{
    n++;
}
```

7.2 Own Function

```
int string_length(string s)
{
    int n = 0;
    while (s[n] != '\0')
    {
        n++;
    }
    return n;
}
```

7.3 strlen

- Provided by <string.h>.
- Example:

  ```
  int length = strlen(s);
  ```

8 ctype.h and Char Operations

8.1 ASCII Insights

- 'A'=65, 'Z'=90, 'a'=97, 'z'=122.
- Difference between 'a' and 'A' is 32.

8.2 Manual Conversions

```
if (s[i] >= 'a' && s[i] <= 'z')
{
    s[i] = s[i] - 32; // to uppercase
}
```

8.3 ctype.h

```
#include <ctype.h>

// e.g.:
s[i] = toupper(s[i]);
// or isalpha(c), islower(c), etc.
```

9 Command-Line Arguments

9.1 Signature of main

```
int main(int argc, string argv[])
{
    ...
}
```

- argc = argument count (words typed)
- argv = array of strings typed by user

9.2 Example Greet (greet.c)

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
```

```
    if (argc == 2)
    {
        printf("hello, %s\n", argv[1]);
    }
    else
    {
        printf("hello, world\n");
    }
}
```

- `./greet David` =¿ ”hello, David”
- `./greet` =¿ ”hello, world”

9.3 Print All Arguments

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    for (int i = 0; i < argc; i++)
    {
        printf("%s\n", argv[i]);
    }
}
```

10 Exit Status

10.1 Returning from `main`

- Return 0 means success, nonzero means error/failure.
- Example:

```
int main(int argc, string argv[])
{
    if (argc != 2)
    {
        printf("Missing arg\n");
        return 1;  // error
    }
    printf("hello, %s\n", argv[1]);
    return 0;     // success
}
```

10.2 Echoing Status

```
./status David
echo $?
```

Prints 0 or 1, etc., depending on `return` from `main`.

11 Cryptography

11.1 Motivation

- **Encryption** scrambles plaintext into ciphertext, reversible if you have the key.
- **Key** is a secret integer or other data that configures the cipher.

11.2 Caesar Cipher

- Shift each alphabetical character by the key.
- e.g. Key=1: ’H’-¿’I’, ’I’-¿’J’. Key=13: *ROT13*.

11.3 Decrypting

- Reverse shift by the same key.
- If ciphertext is e.g. `V Y Y`..., subtract the key from each letter to restore plaintext.

12 Full Code Examples

This section includes *longer* code examples from the lecture:

12.1 `uppercase.c`

```
#include <cs50.h>
#include <ctype.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // Get string
    string s = get_string("Before: ");

    // Print heading
    printf("After:  ");

    // Iterate over string
    for (int i = 0, n = strlen(s); i < n; i++)
    {
        // Convert each char to uppercase
        printf("%c", toupper(s[i]));
    }

    // Move cursor to new line
    printf("\n");
}
```

12.2 length.c

```
#include <cs50.h>
#include <stdio.h>
#include <string.h>

int main(void)
{
    // Prompt user for name
    string name = get_string("Name: ");

    // Use strlen to measure
    int length = strlen(name);
    printf("%i\n", length);
}
```

12.3 greet.c (Two Approaches)

Approach 1 (Using get_string):

```
#include <cs50.h>
#include <stdio.h>

int main(void)
{
    string answer = get_string("What's your name? ");
    printf("hello, %s\n", answer);
}
```

Approach 2 (Using command-line args):

```
#include <cs50.h>
#include <stdio.h>

int main(int argc, string argv[])
{
    if (argc == 2)
    {
        printf("hello, %s\n", argv[1]);
    }
    else
    {
        printf("hello, world\n");
    }
}
```

12.4 status.c

```
#include <cs50.h>
#include <stdio.h>
```

```
int main(int argc, string argv[])
{
    // If user doesn’t type exactly one argument
    if (argc != 2)
    {
        printf("Missing command-line argument\n");
        return 1;   // signal error
    }

    // Otherwise
    printf("hello, %s\n", argv[1]);
    return 0;       // success
}
```

13 Summary and Takeaways

- We explored the entire **compilation** pipeline (preprocessing, compiling, assembling, linking).
- We introduced **debugging** methods:
  - **Rubber duck debugging** (talking it out)
  - `printf` debugging
  - The `debug50` tool with breakpoints
- We introduced the **array** data structure for storing multiple values contiguously.
- **Strings** in C are arrays of `char` plus a `\0` sentinel.
- We used `<string.h>` library functions like `strlen` and `ctype.h` for character classification/conversion.
- **Command-line arguments** (`argc`, `argv`) allow flexible user input from the shell.
- **Exit statuses** (returning 0 or a non-zero integer) let programs communicate success/failure.
- **Cryptography** (e.g. Caesar cipher) demonstrates ASCII-based transformations for encryption/decryption.