Characters and C Programming

ASCII (American Standard Code for Information Interchange)
- specifies encoding (an integer 0..127) for 128 characters, including:
  - 26 lower case letters a-z plus 26 upper case letters A-Z
  - 10 digits 0-9
  - punctuation and other symbols, e.g.: ! @ # $ % ˆ & * ( )
  - some non-printing characters: e.g. newline and space.
- a few languages use only ASCII characters
  - e.g. English, Indonesian
- a few languages use mainly ASCII characters:
  - e.g. German, French, Spanish, Italian, Swedish, Tagalog, Swahili
- many languages use other characters
  - e.g Thai, Greek, Hanzi, Kanji, Korean, Arabic, Cyrillic

Unicode
- specifies encoding for 40,000+ characters
- including almost all languages
- Unicode not covered in COMP1511 (wait for COMP1521)
- ASCII is subset of Unicode

ASCII in C Programs
- ASCII specifies mapping of 128 characters to integers.
- You don’t have to memorize ASCII codes
- In C single quotes give you the ASCII code for a character:
  - printf("%d\n", 'a'); // prints 97
  - printf("%d\n", 'A'); // prints 65
  - printf("%d\n", '0'); // prints 48
  - printf("%d,%d\n", ' ', '\n'); // prints 32,10

  - Don’t put ASCII codes in your program – use single quotes instead.

Writing a Character - putchar or printf %c
- putchar writes a byte to standard output.
- %c in printf format does same thing

```c
int c1 = 'a';
int c2 = '\n';
putchar(c1); // prints 'a'
putchar(c2); // prints '\n'
putchar('a'); // prints 'a'
putchar('\n'); // prints '\n'
printf("%c\n", c); // prints 'a' and '\n'
putchar(97); // prints 'a' but DON'T DO THIS
putchar(10); // prints '\n' but DON'T DO THIS
```

Printing All ASCII Characters
```c
int ascii = 33;
while (ascii < 127) {
  printf("In ASCII %d prints as %c\n", ascii, ascii);
  ascii = ascii + 1;
}
```

```
$ dcc ascii.c -o ./ascii
$ ./ascii
In ASCII 33 prints as !
In ASCII 34 prints as "
In ASCII 35 prints as #
In ASCII 36 prints as $
In ASCII 37 prints as %
...
In ASCII 124 prints as |
In ASCII 125 prints as }
In ASCII 126 prints as ~
```
Manipulating Characters

The ASCII codes for the digits, the upper case letters and lower case letters are contiguous. This allows some simple programming patterns:

```c
// check for lowercase
if (c >= 'a' && c <= 'z') {
    ...
}
```

```c
// check is a digit
if (c >= '0' && c <= '9') {
    // convert ASCII code to corresponding integer
    numeric_value = c - '0';
}
```

Reading a Character - getchar

C provides library functions for reading and writing characters

- `getchar` reads a byte from standard input.
  - `getchar` returns an `int`
    - a successful `getchar` returns a value 0 .. 255
    - if input is from a terminal this is likely be an ASCII code 0..127
    - if it can not read a byte `getchar` returns a special value `EOF` (usually -1)
  - Otherwise `getchar` returns an integer (0..255) inclusive.
  - Beware input often buffered until entire line can be read.

```c
printf("Please enter a character: ");
int c = getchar();
printf("The ASCII code of the character is %d
", c);
```

Consider the following code:

```c
printf("Please enter first character:
");
int c1 = getchar();
printf("Please enter second character:
");
int c2 = getchar();
printf("First: %c
Second: %c
", c1, c2);
```

The newline character from pressing Enter will be the second character read.

How can we fix the program?

```c
printf("Please enter first character:
");
int c1 = getchar();
getchar(); // reads and discards a character
printf("Please enter second character:
");
int c2 = getchar();
printf("First: %c
Second: %c
", c1, c2);
```
Input functions such as `scanf` or `getchar` can fail because no input is available, e.g., if input is coming from a file and the end of the file is reached. On UNIX-like systems (Linux/OSX) typing `Ctrl + D` signals to the operating system no more input from the terminal. Windows has no equivalent - some Windows programs interpret `Ctrl + Z` similarly. `getchar` returns a special value `EOF` to indicate there is no input was available. This non-ASCII value is `#`defined as `EOF` in `stdio.h`. On most systems `EOF` == -1. There is no end-of-file character (on modern operating systems).

### Programming pattern for reading characters to the end of input:

```c
int ch = getchar();
while (ch != EOF) {
    printf("%c read, ASCII code is %d\n", ch, ch);
    ch = getchar();
}
```

Source code for `getchar_eof.c`.

For comparison a similar programming pattern for reading integers

```c
int num;
while (scanf("%d", &num) == 1) {
    printf("you entered the number: %d\n", num);
}
```

Source code for `getchar_eof.c`.

### A Function that Converts Characters to Upper Case

```c
int make_upper_case(int character) {
    if (character >= 'a' && character <= 'z') {
        int alphabetPosition = character - 'a';
        return 'A' + alphabetPosition;
    } else {
        return character;
    }
}
```

Source code for `upper_case.c`.

### Use A Function that Converts Characters to Upper Case

```c
int main(void) {
    int character = getchar();
    while (character != EOF) {
        int new_character = make_upper_case(character);
        putchar(new_character);
        character = getchar();
    }
    return 0;
}
```

Source code for `upper_case.c`.
Useful C Library Functions for Characters

The C library includes some useful functions which operate on characters. Several of the more useful listed below.

```c
#include <ctype.h>

int toupper(int c); // convert c to upper case
int tolower(int c); // convert c to lower case
int isalpha(int c); // test if c is a letter
int isdigit(int c); // test if c is a digit
int islower(int c); // test if c is lower case letter
int isupper(int c); // test if c is upper case letter
```

The char Type

- The C type char stores small integers.
  - 8 bits (almost always).
  - always can hold integers 0 .. +127.
  - char mostly used to store ASCII character codes.
- In 1511 only use char for arrays of character codes
  - Don’t use char for individual variables
- Even if a numeric variable is only used for the values 0..9,
  - still use the type int for the variable.

```c
// works but DO NOT do this, use int
char i = 6;
char j = 7;
char k = i * j;
printf("%d
", k); // prints 42
```

What is A String

- string in computer science means a sequence of characters.
  - in other words, when we have multiple characters together, we call it a string
- In C strings are an array of char variables containing ASCII codes.
- These arrays have an extra last element containing a 0
  - this 0 marks the end of the string
  - The extra 0 can also be written ‘\0’
  - often gets called a null character or null-terminator.
  - very convenient - programs don’t have to track the length of the string.
  - 0 in ASCII reserved for this purpose
  - no actual character has ASCII code of 0

Strings

C provides a convenient syntax shorthand (") which makes strings more readable.

All of these 3 declarations do exactly the same thing.

```c
// BAD: DON'T DO THIS - using ASCII code is completely unreadable
char word0[] = { 104, 101, 108, 108, 111, 0 };

// OK: but not easy to read or type
char word1[] = { 'h', 'e', 'l', 'l', 'o', '0' };

// we also have a convenient shorthand
// GOOD - much easier to read
char word2[] = "hello";
```
8 ways to Print A Message - #0

// DO NOT PUT ASCII CODES IN YOUR PROGRAM LIKE THIS
putchar(65);  // printf("%c", 65) is equivalent
putchar(110);
putchar(100);
putchar(114);
putchar(101);
putchar(119);
putchar(32);
putchar(82);
putchar(111);
putchar(99);
putchar(107);
putchar(115);
putchar(33);
putchar(10);

source code for andrew_rocks0.c

8 ways to Print A Message - #1

putchar('A');  // equivalent to putchar(65) but readable!
putchar('n');
putchar('d');
putchar('r');
putchar('e');
putchar('w');
putchar(' ');
putchar('R');
putchar('o');
putchar('c');
putchar('k');
putchar('s');
putchar('
');

source code for andrew_rocks1.c

8 ways to Print A Message - #2

int asciiCodes[14];
asciiCodes[0] = 'A';
asciiCodes[1] = 'n';
asciiCodes[2] = 'd';
asciiCodes[3] = 'r';
asciiCodes[4] = 'e';
asciiCodes[5] = 'w';
asciiCodes[6] = ' ';
asciiCodes[7] = 'R';
asciiCodes[8] = 'o';
asciiCodes[9] = 'c';
asciiCodes[10] = 'k';
asciiCodes[11] = 's';
asciiCodes[12] = '!';
asciiCodes[13] = '
';

int i = 0;
while (i < 14) {
    putchar(asciiCodes[i]);
    i = i + 1;
}

source code for andrew_rocks2.c

8 ways to Print A Message - #3

// we can put all the ASCII codes in array and
// then print every array element
// we need to track the array size (should be a #define)
int ascii_codes[14] = { 'A', 'n', 'd', 'r', 'e', 'w', ' ',
                        'R', 'o', 'c', 'k', 's', '!', '
'};

int i = 0;
while (i < 14) {
    putchar(ascii_codes[i]);
    i = i + 1;
}

source code for andrew_rocks3.c
8 ways to Print A Message - #4

// we can put all the ASCII codes in array
// if we put a special value (0) as last array element
// We don't need to track the array size which is more convenient.
// We don't specify the size of an array being initialized so
// C makes it big enough to hold all the v (15 in this case)
int ascii_codes[] = {'A', 'n', 'd', 'r', 'e', 'w', ' ', 'R', 'o', 'c', 'k', 's', '!', '\n', 0};

int i = 0;
while (ascii_codes[i] != 0) {
    putchar(ascii_codes[i]);
    i = i + 1;
}
source code for andrew_rocks4.c

8 ways to Print A Message - #5

// same as last version except use a char array
char ascii_codes[] = {'A', 'n', 'd', 'r', 'e', 'w', ' ', 'R', 'o', 'c', 'k', 's', '!', '\n', 0};

int i = 0;
while (ascii_codes[i] != 0) {
    putchar(ascii_codes[i]);
    i = i + 1;
}
source code for andrew_rocks5.c

8 ways to Print A Message - #6 & #7

// C has a shorthand for char arrays containing a sequence of
// ASCII codes with an extra 0 value marking the end
// Its "Andrew Rocks!"
char ascii_codes[] = "Andrew Rocks!\n";

int i = 0;
while (ascii_codes[i] != 0) {
    putchar(ascii_codes[i]);
    i = i + 1;
}
source code for andrew_rocks6.c

// Many C library functions work with zero-terminated char arrays
// puts prints a zero-terminated char array
fputs(asciiCodes, stdout);
// printf "%s" also prints a zero-terminated char array
printf("%s", asciiCodes);
source code for andrew_rocks7.c
Command Line Arguments

Sometimes we want to give information to our program at the moment when we run it.

The "Command Line" is where we type in commands into the terminal.

Arguments are another word for input parameters.

This extra text we type after the name of our program can be passed into our program as strings.

$./my_program extra information 1 2 3

Main functions that accept arguments

- So far we’ve used a main function that looks like this:

```c
int main(void) {
}
```

- There is a second definition we can use for `main`:

```c
int main(int argc, char *argv[]) {
}
```

- `argc` will contain an "argument count"
  - This will be an integer of the number of words that were typed in (including the program name)
- `argv` will be the "argument values"
  - This will be an array of strings where each string is one of the words

Accessing Command Line Arguments

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
    printf("argc = %d\n", argc);
    printf("My name is %s\n", argv[0]);
    // print command line arguments
    int i = 1;
    while (i < argc) {
        printf("Argument %d is: %s\n", i, argv[i]);
        i = i + 1;
    }
    return 0;
}
```

Source code for `print_arguments.c`

Converting Command-line Arguments To Numbers

`stdlib.h` defines useful functions to convert strings.

- `atoi` converts string to integer
- `strtol` also converts string to integer
  - more complicated to call
  - handles other bases (e.g. hexadecimal)
  - has error checking
- `atof` converts string to double
  - more complicated to call
  - handles other bases (e.g. hexadecimal)
  - has error checking
Example - Converting Command-line Arguments To Numbers

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    int sum = 0;
    int argument = 1;
    while (argument < argc) {
        // convert string to integer
        int n = atoi(argv[argument]);
        // you could use `strtol` instead of `atoi` like this
        // int n = strtol(argv[argument], NULL, 10);
        sum = sum + n;
        argument = argument + 1;
    }
    printf("Total is %d\n", sum);
    return 0;
}
```

Do 2 command-line Arguments rhyme? - main

```c
int main(int argc, char *argv[]) {
    // check we have two command-line arguments
    if (argc != 2) {
        printf("Usage %s <word1> <word2>\n", argv[0]);
        return 1;
    }
    // calculate their rhyming score
    int score = simple_rhyming_count(argv[1], argv[2]);
    printf("Rhyming score is %d\n", score);
    return 0;
}
```

Do 2 command-line Arguments rhyme? - simple calculation

```c
int simple_rhyming_count(char word1[], char word2[]) {
    int r_count = 0;
    int i = 0;
    while (word1[i] != 0 && word2[i] != 0) {
        // if same ASCII code in same position
        if (word1[i] == word2[i]) {
            r_count++;
        }
        i++;
    }
    return r_count;
}
```

Do 2 command-line Arguments rhyme? - complex calculation

```c
int rhyming_count(char word1[], char word2[]) {
    int r_count = 0;
    int i = 0;
    while (word1[i] != '0') {
        // if same ASCII code in same position
        if (letter_is_anywhere_in_word(word1[i], word2)) {
            r_count++;
        }
        i++;
    }
    return r_count;
}
```

source code for sum_arguments.c

source code for rhyme_simple.c

source code for rhyme.c
Do 2 command-line Arguments rhyme? - check for letter
// return 1 if letter occurs anywhere in word
// return 0, otherwise
// e.g letter_is_anywhere_in_word('e', "leet") returns 1
// letter_is_anywhere_in_word('k', "cat") returns 0
int letter_is_anywhere_in_word(int letter, char word[]) {
    int i = 0;
    while (word[i] != '\0') {
        if (word[i] == letter) {
            return 1;
        }
        i++;
    }
    // didn't find letter anywhere in word
    return 0;
}

fgets - Read a Line
• fgets(array, array_size, stream) reads a line of text
• array - char array in which to store the line
• array_size - the size of the array
• stream - where to read the line from, e.g. stdin
• fgets will not store more than array_size characters in array
• fgets always stores a ' 0' terminating character in the array.
• fgets stores a ' n' in the array if it reads entire line
  • often need to overwrite this newline character
int i = strlen(line);
if (i > 0 && line[i - 1] == '\n') {
    line[i - 1] = '\0';
}

Reading an Entire Input Line
You might use fgets as follows:
#define MAX_LINE_LENGTH 1024
...
char line[MAX_LINE_LENGTH];
printf("Enter a line: ");
// fgets returns NULL if it can't read any characters
if (fgets(line, MAX_LINE_LENGTH, stdin) != NULL {  
    fputs(line, stdout);
    // or
    printf("%s", line); // same as fputs
}

Reading Lines to End of Input
Programming pattern for reading lines to end of input:
// fgets returns NULL if it can't read any characters

while (fgets(line, MAX_LINE, stdin) != NULL) {
    printf("you entered the line: %s", line);
}
Do 2 words from stdin Arguments rhyme?

```c
int main(void) {
    char first_word[MAX_WORD_LENGTH];
    printf("Enter first word: ");
    fgets(first_word, MAX_WORD_LENGTH, stdin);
    char second_word[MAX_WORD_LENGTH];
    printf("Enter second word: ");
    fgets(second_word, MAX_WORD_LENGTH, stdin);
    int score = rhyming_count(first_word, second_word);
    printf("Rhyming score is %d\n", score);
    return 0;
}
```

Example - detect identical consecutive lines - broken

- We can't compare or assign arrays like we do with ints and doubles
  ```c
  char line[MAX_LINE];
  char last_line[MAX_LINE];
  printf("Enter line: ");
  fgets(last_line, MAX_LINE, stdin);
  printf("Enter line: ");
  while (fgets(line, MAX_LINE, stdin) != NULL) {
    // DOES NOT WORK - arrays can not be compared
    if (line == last_line) {
      // lines are identical
      printf("Snap!\n");
    }
    // DOES NOT WORK - arrays can not be assigned
    last_line = line;
    printf("Enter line: ");
  }
  ```

Example - comparing zero-terminated char arrays

// return 1 if array1 & array2 differ in any element, 0 otherwise
// array1 & array2 must be null-terminated char arrays
// strcmp from <string.h> performs similar function
```c
int compare_arrays(char array1[], char array2[]) {
    int i = 0;
    while (array1[i] != \0) {
        if (array1[i] != array2[i]) {
            return 1;
        }
        i = i + 1;
    }
    if (array2[i] == \0) {
        return 0;
    } else {
        return 1;
    }
}
```

Example - assigning zero-terminated char arrays

// copy elements in source_array to destination_array
// source_array must be a null-terminated char array
// destination_array must be large enough to hold string
// strcpy from <string.h> performs the same function
```c
void copy_array(char destination_array[], char source_array[]) {
    int i = 0;
    while (source_array[i] != \0) {
        destination_array[i] = source_array[i];
        i = i + 1;
    }
    destination_array[i] = \0;
}
```

Example - comparing zero-terminated char arrays

// return 1 if array1 & array2 differ in any element, 0 otherwise
// array1 & array2 must be null-terminated char arrays
// strcmp from <string.h> performs similar function
```c
int compare_arrays(char array1[], char array2[]) {
    int i = 0;
    while (array1[i] != \0) {
        if (array1[i] != array2[i]) {
            return 1;
        }
        i = i + 1;
    }
    if (array2[i] == \0) {
        return 0;
    } else {
        return 1;
    }
}
```

Example - assigning zero-terminated char arrays

// copy elements in source_array to destination_array
// source_array must be a null-terminated char array
// destination_array must be large enough to hold string
// strcpy from <string.h> performs the same function
```c
void copy_array(char destination_array[], char source_array[]) {
    int i = 0;
    while (source_array[i] != \0) {
        destination_array[i] = source_array[i];
        i = i + 1;
    }
    destination_array[i] = \0;
}
```

Example - comparing zero-terminated char arrays

// return 1 if array1 & array2 differ in any element, 0 otherwise
// array1 & array2 must be null-terminated char arrays
// strcmp from <string.h> performs similar function
```c
int compare_arrays(char array1[], char array2[]) {
    int i = 0;
    while (array1[i] != \0) {
        if (array1[i] != array2[i]) {
            return 1;
        }
        i = i + 1;
    }
    if (array2[i] == \0) {
        return 0;
    } else {
        return 1;
    }
}
```

Example - assigning zero-terminated char arrays

// copy elements in source_array to destination_array
// source_array must be a null-terminated char array
// destination_array must be large enough to hold string
// strcpy from <string.h> performs the same function
```c
void copy_array(char destination_array[], char source_array[]) {
    int i = 0;
    while (source_array[i] != \0) {
        destination_array[i] = source_array[i];
        i = i + 1;
    }
    destination_array[i] = \0;
}
```
Example - detect identical consecutive lines - working

```c
char line[MAX_LINE];
char last_line[MAX_LINE];
// read first line into array last_line
printf("Enter line: ");
fgets(last_line, MAX_LINE, stdin);
printf("Enter line: ");
while (fgets(line, MAX_LINE, stdin) != NULL) {
    if (compare_arrays(line, last_line) == 0) {
        // lines are identical
        printf("Snap!\n");
    }
    copy_array(last_line, line);
    printf("Enter line: ");
}
```

source code for snap_line_ok.c

Example - detect identical consecutive lines - better

```c
string.h provides functions for comparing and assigning zero-terminated char arrays

char line[MAX_LINE];
char lastLine[MAX_LINE];
// read first line into array lastLine
printf("Enter line: ");
fgets(lastLine, MAX_LINE, stdin);
printf("Enter line: ");
while (fgets(line, MAX_LINE, stdin) != NULL) {
    if (strcmp(line, lastLine) == 0) {
        // lines are identical
        printf("Snap!\n");
    }
    strncpy(lastLine, line, MAX_LINE);
    printf("Enter line: ");
}
```

source code for snap_line_good.c

string.h - a quick summary of useful functions

```c
#include <string.h>
// length of s - does not count the '\0'
int strlen(char *s);
// compare s1 & s2 - return 0 iff they are identical
int strcmp(char *s1, char *s2);
// copy contents of src to dest
char *strcpy(char *dest, char *src);
// copy contents of src to dest but not more than n elements
char *strncpy(char *dest, char *src, int n);
// append contents of src to dest
char *strcat(char *dest, char *src);
// append contents of src to dest but not more than n elements
char *strncat(char *dest, char *src, int n);
// search for c in s, return pointer to first occurrence
char *strchr(char *s, int c);
// search for c in s, return pointer to last occurrence
char *strrchr(char *s, int c);
```