Reference Sheet for CO120.3 Programming III

Summer 2017

Basic Types

1. Don't use the `char`, `short`, `int` and `long` types.

In order to increase portability and readability, use `stdint.h`.
Included types are: `uint8_t`, `uint16_t`, `uint32_t`, `uint64_t`, `int8_t`, `int16_t`, `int32_t`, `int64_t`.

Enumerated Types Example:

```
/**
 * @brief An enum type that represents flags for rendering.
 *
 * Each bit represents a different flag. Use bitwise and
 * to check if a flag is set.
 */
enum render_flag {
  /** The ambient flag (bit 0). */
  AMBIENT = 1,
  /** The diffuse flag (bit 1). */
  DIFFUSE = 2,
  /** The specular flag (bit 2). */
  SPECULAR = 4,
};
```

Functions

1. Use underscore_separation for function names.
2. Don't put a space between the function name and parameter list.
3. Never leave the parameter list for a function empty.
4. Leave a single space between the closing parenthesis of the parameter list and the opening brace of the function body.

Note: Try to keep functions under 40 lines long.
Remember to include `void` in your parameter list for functions without parameters.

Variables Note: In general, you should initialise variables at point of definition.
Example: `double numbers[2][3] = {{1.0, 2.0, 3.0}, {2.0, 3.0, 4.0}}`.
A string can be represented as an array of type `char`, ending with `'/0'`.
Example: `uint8_t hello[] = "Hello!"`.

Flow of Control

1. Always use braces where it is possible to do so.
2. Leave a space and after the opening and closing parentheses for the condition, respectively.

Note: You should usually break at the end of each case in a switch statement.
Examples:

```
if (condition) {
  statement;
  ...
} else if (condition) {
  ...
} else {
  ...
}

while (condition) {
  ...
}

for (initialiser; condition; increment / decrement) {
  ...
}
```

```
switch (expression) {
  case constant1:
    ...
    break;
  case constant2:
    ...
    break;
  default:
    assert(false);
}
```

Logical Expressions

1. Place constants before variables when using the equality operator.

Note: Use parentheses to clarify precedence in complex expressions.
Example: `3 == x`, NOT `x == 3`.

Input Output Use `printf` to print to standard output. Sanitise using: - `%c` for character. - `%d` for signed integer. - `%u` for unsigned integer. - `%o` for octal. - `%x` for hexadecimal. - `%f` for floating point value. - `%e` for floating point value using scientific notation. - `%s` for string. - `%p` for pointer value.

Use `scanf` to read from standard input. Beware of secuirity risks when inputting, say, a string. You can use `fgets` instead.
Examples:

```
int i;
int ret = scanf("%i", &i);
assert(1 == ret);
```

```
char buffer[100];
int size = sizeof(buffer);
fgets(buffer, size, stdin);
```

Bitwise Operations

- `&` for bitwise AND.
- `|` for bitwise OR.
- `>>` for RIGHT SHIFT.
- `<<` for LEFT SHIFT.
- `~` for bitwise NOT.
- `^` for bitwise XOR.

Pointers

1. When declaring a pointer, place the `*` adjacent to the variable name, not the type.
2. When passing by reference, pointers must be declared as `const`.

Example: `int *pointer`, NOT `int* pointer` or `int * pointer`.
Note: Use the `const` keyword wherever possible.
Examples:

```
int val = 5;
// Value cannot be modified
const int *ptr1 = &val;
// Pointer cannot be modified
int const *ptr2 = &val;
// Neither value nor pointer can be modified
const int const *ptr3 = &val;
```

Think of `*` as an operator that takes an address and returns the value which it points to. `&` is an operator that takes a value and returns its address.
We can also use function pointers to pass functions be reference.
Example:

```
/**
 * @brief A function which sums its two inputs.
 */
int sum(int a, int b) {
  return a + b;
}

/**
 * @brief A main function which does nothing.
 */
int main(void) {
  int (*sum_pointer)(int, int) = &sum;

  return EXIT_SUCCESS;
}
```

Command Line Arguments `main` can have a type signature where it receives arguments from the command line:

- `argc`, the number of passed parameters.
- `argv`, an array of strings.

The first argument is the name of the file!

Example:

```
/**
 * @brief A main function which prints its command line arguments.
 */
int main(int argc, char **argv) {
  for (int i = 0; i < argc; i++) {
    printf("argv[%i] = %s\n", i, argv[i]);
  }
  return EXIT_SUCCESS;
}
```

Memory Management

1. `main` should only return `EXIT_SUCCESS` or `EXIT_FAILURE`.

- `stlib.h`'s `void *malloc(size_t size)` allocates a region of memory of `size` bytes and returns a pointer to the allocated memory.
- `void free(void *ptr)` frees a previously allocated memory region.
- You need to check that these don't fail (not NULL).
- `string.h`'s `void *memset (void *s, int c, size_t n)` sets teh `n`-byte region starting at `s` to `c`.
- `void *memcpy(void *dest, const void *src, size_t n)` copies `n` bytes from `src` to `dst`, returning `dst`.
- You can use the `memcheck` tool provided by `valgrind` to check for memory leaks.

Assertions `assert.h` provides `assert(logical_expression)`.

Headers

1. All headers should be surrounded by include guards, `#ifndef THIS_H`, `#define THIS_H`.

Note: In general, every `.c` file should have an associated `.h` file.

Makefile Example:

```
# Set the flags for the C compiler
CFLAGS = -Wall -pedantic -std=c99

# Build rule for final executable
sum: add.o

# Build rules for the .o files
sum.o: add.h
add.o: add.h

# Rule to clean generated files
clean: rm -f sum sum.o add.o

# Tell make that 'clean' is not a real file
.PHONY: clean
```

More Variables

- `static` (local): value is retained.
- `static` (top level): only seen within this file.
- `extern` variables: defined in header, can be used in multiple files.
- Global variables: try to avoid them.

File Input

- Use `FILE fopen(const char *path, const char *mode)`.
- Modes are `r` to read from start, `w` to write from start, `a` to append.
- Need to check file could be opened (not NULL).
- Use `int feof(FILE *stream)` to check if end of file has been reached (NULL).
- Use `int fclose(FILE *fp)` to close file.
- Need to check it doesn't fail.

Reading a file:

- `int fscanf(FILE *stream, const char *format, ...)`.
- `char *fgets(char *s, int size, FILE *stream)`.

Writing to a file:

- `int fprintf(FILE *stream, const char *format, ...)`.
- `int fputs(const char *s, FILE *stream)`.

Binary data:

- Use the mode `b`.
- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream)`
- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream)`
- `ptr` is the address to read or write, `nmemb` is the number of elements, `size` is the number of bytes for each element.
- Need to check returned value is the number of elements.

Error checking:
You can also use `int ferror(FILE *stream)`.
Printing errors:

- Use `perror(char *msg)` when printing non user defined errors.
- Otherwise, use `fprintf(stderr, char *msg)`.

Structs Example:

```
/**
 * @brief A struct that hold the co-ordinates of a 2D vertex.
 */
typedef struct {
  /** The x co-ordinate */
  float x;
  /** The y co-ordinate */
  float y;
} vertex;
```

- Use `.` to refer to a struct component.
- Use `->` to refer to a struct component from a pointer to a struct.

Use `typedef` to declare an alias for an existing type.
Use a `union` where requirements to store components are mutually exclusive.

Constants Three approaches:

1. Use a marco using `#define`.
2. Define a global `static const` variable (not a true constant).
3. Use an unnamed `enum`.

Strings Use `string.h` to get:

- `size_t strlen(const char *s)`.
- `char *strncat(char *dest, const char *src, size_t n)`.
- `int strncmp(const char *s1, const char *s2, size_t n)`.
- `char *strncpy(char *dest, const char *src, size_t n)`.

Tips for Lexis Tests Make sure you check for all errors and initialise variables immediately. In particular:

- All functions with a pointer as an argument should check that the given pointer is not NULL.
- Any memory that you malloc must be checked not NULL and should be initialised immediately to 0.