# Assignment: Introduction to C

## Learning Outcomes

- Practice functional decomposition and algorithm design
- Apply standard C library functions to solve text-based problems
- Parse and understand behavior of C code

## Introduction

You're required to print textual art to `stdout` (that is, the standard output which by default is your computer's screen) by defining functions `draw_tree` and `draw_animal` in source file `q.c`. A driver file `qdriver.c` with the definition of function `main` containing calls to functions `draw_tree` and `draw_animal` is provided:

```c
// prototypes for functions called in function main
#include <stdio.h> // for printf

void draw_tree(void);   // defined by student in q.c
void draw_animal(void); // defined by student in q.c

int main(void) {
  draw_tree();
  draw_animal();
  return 0;
}

```

To test correctness of your definitions, text file `output.txt` containing the correct output written to `stdout` by the two functions is provided. Note that you will not be submitting `qdriver.c`. The submission page has its own copy of `qdriver.c` exactly similar to the one described in the code block shown above. The automatic grader will compile your `q.c`, then link with both `qdriver.o` and with standard C library to create an executable. The automatic grader will determine the submission grade by running the executable and comparing its output with the correct output in `output.txt`.

### File-level documentation

Open a Window command prompt, change your directory to `C:\sandbox`, create a directory `ass02` (if it doesn't exist), and launch the Linux shell. Using Code, create a new source file called `q.c`. Begin by inserting a *file-level* documentation block at the top of source file `q.c`. Every source and header file you submit *must* contain file-level documentation whose purpose is to provide human readers (yourself and other programmers) useful information about the purpose of this source file at some later point of time (could be weeks later or months later or even years later). Here is a *template* of a file-level documentation block:

```
/*!
@file     @todo what is name of this source file?
@author   @todo provide your name & DP login: Nicolas Pepe (nicolas.pepe)
@course   @todo which course is this source file meant for?
@section  @todo which section of this course are you enrolled in?
@tutorial @todo provide Tutorial #
@date     @todo provide date on which you created the file
@brief    @todo provide a brief explanation about what this source file
          does like the example description below:
          This file contains a collection of functions that puts a salmon
          on a cedar plank and smokes the fish for three hours. Remove the
          plank with the fish, throw away the fish, and enjoy the plank.
*//*_____________________________________________________________________*/
```

Make modifications to the template by replacing `@todo` with your information. Providing a file header is mandatory for any submissions you make in this course.

## Function prototypes and including `stdio.h`

Since functions `draw_tree` and `draw_animal` must write characters to `stdout`, you'll be calling either function `printf` or function `puts`, both of which are defined in the standard C library. From lectures, you've learnt that anytime identifiers such as `printf` are used in code, the identifier must be *declared* to the compiler. For functions, this declaration consists of a *function prototype*. Recall that a function prototype is a declaration that introduces the function's name, parameter list, and return type. The function prototype for standard C library function `printf` is located in header file `stdio.h` which is supplied by the vendor of the C compiler. Therefore, you've to include header file `stdio.h` in `q.c` using preprocessor `include` directive:

```
#include <stdio.h> // for printf
```

## Function stubs

You don't want to completely author definitions of functions `draw_tree` and `draw_animal` before initiating the edit-compile-link-run cycle. Instead, you want to run this edit-compile-link-run cycle frequently whenever a few lines of code are added so that you can spot syntax errors and bugs as early as possible. Introduce stubs (that is, empty function definitions) of functions `draw_tree` and `draw_animal` with temporary calls to function `printf`, like this:

```
void draw_tree(void) {
  printf("this is draw_tree\n");
}

void draw_animal(void) {
  // equivalently standard library function puts can be
  // used to write text to standard output
  puts("this is draw_animal\n");
}

```

### Compile-link-run cycle

Now, you're ready to compile and link the two source files:

```
digipen@dit0701sg:/mnt/c/sandbox/tut02$ gcc -std=c11 -pedantic-errors -
Wstrict-prototypes -Wall -Wextra -Werror qdriver.c q.c -o q.out
digipen@dit0701sg:/mnt/c/sandbox/tut02$
```

If there are any compile or linker errors, read the diagnostic messages from the compiler to identify the source file(s) emitting these message(s).

Assuming clean compilation without diagnostic messages, run executable `q.out` to ensure the expected characters are written to `stdout`:

```
digipen@dit0701sg:/mnt/c/sandbox/tut02$ ./q.out
this is draw_tree
this is draw_animal
digipen@dit0701sg:/mnt/c/sandbox/tut02$
```

## Task 1: Definition of `draw_tree`

### What should the function do?

In source file `q.c`, define a function `draw_tree` that writes to `stdout` the following textual art consisting of $9$ lines that represent a tree:

```
    *
   ***
  *****
 *******
*********
    #
    #
    #
    #

```

### Function-level documentation

Every function that you define and submit for assessment must contain *function-level documentation*. This documentation should consist of a description of the function, the inputs, and return value. This will allow you and other team members to quickly grasp the details necessary to efficiently debug and maintain the function. Certain details that programmers find useful include: what does the function take as input, what is the output, a sample output for some example input data, how the function implements its task, and importantly any special considerations that the author has taken into account in implementing the function. Although beginner programmers might feel that these details are unnecessary and are an overkill for assignments, they have been shown to save considerable time and effort in both academic and professional settings. Humans are prone to quickly forget details and good function-level documentation provides continuity for developers by acting as a repository for information related to the function. Otherwise, the developer will have to unnecessarily invest time in recalling and remembering undocumented gotcha details and assumptions each time the function is debugged or extended to incorporate additional features. An example of function-level

documentation is provided below for function `reverse32()` that returns the byte order of an unsigned 32-bit integer.

```
/*!
@brief Reverses the byte order of an unsigned 32-bit integral value.

This function takes as input a 32-bit unsigned integer and reverses
the byte order of the full word. That is, given the value 0x12345678,
the function will return value 0x87654321.
Special scenarios: This function assumes the parameter is of type
unsigned int. Behavior is unspecified if the parameter is signed.

@param word - the 32-bit unsigned int to be reversed.
@return the reversed 32-bit unsigned int.
*//*_____________________________________________________________________*/
unsigned int reverse32(unsigned int word);
```

Make modifications to the above sample with information specific to function `draw_tree` and place this function-level documentation above the definition of function `draw_tree`.

## Implementation

Consider your stub definition of function `draw_tree`:

```
void draw_tree(void) {
  printf("this is draw_tree\n");
}
```

It is clear that the only thing that you need to do to generate the expected output is to add 9 statements with each statement containing a call to function `printf` (or function `puts`). Recall from class lectures that function `printf` is not a part of the core C language. Instead, it is a part of the standard C library and is implemented by the developers of the `gcc` compiler toolchain. Here are some details about function `printf`:

1. Function `printf` is used to print characters to `stdout` (the *print* in `printf` hints at printing something to `stdout`).
2. The left and right parentheses following identifier `printf` tell the compiler that `printf` is a function. This function is defined in the standard C library and is used to print formatted data to `stdout`. Before calling `printf` to print to `stdout`, you must provide the prototype (or declaration) for the function. This is ensured by including header file `stdio.h` in file `q.c` using the preprocessor `include` directive.
3. Between the left and right parentheses, the text enclosed in double quotes `"this is draw_tree\n"` is called a string literal. A *string literal* is a series of characters delimited by pair of double quotes `"`.
4. String literal `"this is draw_tree\n"` is the *argument* or *value* passed to function `printf`. The string literal passed as argument to function `printf` is called the *format string* because it specifies how characters are to be printed to `stdout`.
5. Ordinary characters in the format string are printed to `stdout` as is. The `%` character is a non-ordinary character and it specifies how characters are to be printed to `stdout`. Since there are no `%` characters in control string `"this is draw_tree\n"`, the sequence of characters `this is draw_tree` are printed to `stdout`.

6. Backslash character `\` in a string literal (or format string) is called the *escape* character. C combines character `\` with the character that follows it and then attaches a special meaning to the combination of two characters. C does this to provide a way for programmers to specify non-printable characters such as a tab, backspace, newline, or to indicate the keyboard's alarm ring. The combination of \ and `n`, as in `\n` is the *newline* escape sequence which causes a skip to a new line on `stdout` after the sequence of characters `this is draw_tree` is printed to `stdout`. This enables characters `this is draw_tree` to be printed to a complete line. The cursor is a moving place marker that indicates the next position in `stdout` (on the screen, for example) where information will be displayed. When executing a call to function `printf`, the cursor is advanced to the start of the next line in `stdout` if the `\n` escape sequence is encountered in the string passed to the function. A `printf` string often ends with a `\n` newline escape sequence so that the call to `printf` produces a completed line of output.
7. Clearly, replacing string literal `"this is draw_tree\n"` with `"    *\n"` will print this text to `stdout` with the cursor located at the start of line $2$:

```
    *

```

   Thus, the first statement will now look like this: `printf("    *\n");`. A second `printf` statement: `printf("   ***\n");` will print this text to `stdout`:

```
   *
  ***

```

8. Further addition of statements with calls to function `printf` with appropriate string literal arguments would generate the required textual art. A total of $9$ statements with each statement containing a call to function `printf` will do the trick. Ensure the output from function `draw_tree` matches the correct output.

# Task 2: Definition of `draw_animal`

## Expected output

Begin authoring the definition of function `draw_animal` by implementing a function-level documentation block. Using expertise gained in defining function `draw_tree`, replace the stub implementation of function `draw_animal` with calls to function `printf` that will print the following text to *stdout*:

```
  /\     /\
 /  \___/  \
(           )    -------
(   '   '   )   / Hello  \
(     _     )  <  Junior  |
(           )   \ Coder! /
 |         |     -------
 |    |    |
 |    |    |
(_____|_____)

```

Notice that the animal's pointy ears require printing the backslash character `\`. Recall that C uses backslash character `\` as an escape character to change the meaning of the next character after it. Therefore, a backslash character `\` cannot be printed using string literal `"\"`:

```
printf("\");
```

This call to function `printf` won't compile because of the malformed string literal argument to `printf`: C will assume that `\"` means `"` because `\` is an escape character resulting in a missing right double quote to terminate the string. Instead, to print a backslash character `\`, it must be escaped with a preceding escape character . A call to function `printf` with string literal `"\\"` will then print a backslash character `\`:

```
printf("\\");
```

To print a newline after the backslash character, string literal `"\\"` must be terminated with the escape sequence `\n`:

```
printf("\\\n");
```

Add the necessary calls to function `printf` to print the exact same set of characters to complete the definition of function `draw_animal` and generate the textual art representing the animal and greeting.

## Compiling, linking, and testing

Compile and link your source file `q.c` along with driver source file `qdriver.c` using the full suite of required `gcc` options:

```
digipen@dit0701sg:/mnt/c/sandbox/tut02$ gcc -std=c11 -pedantic-errors -
Wstrict-prototypes -Wall -Wextra -Werror qdriver.c q.c -o q.out
```

Run executable `q.out` like this to display the characters on `stdout`:

```
digipen@dit0701sg:/mnt/c/sandbox/tut02$ ./q.out
```

Using the shell's redirection operator `>` (you learnt about redirection in Lab 1):

```
digipen@dit0701sg:/mnt/c/sandbox/tut02$ ./q.out > your-output.txt
```

You're given output file `output.txt` containing the correct output. Your output which was redirected to file `your-output.txt` must exactly match the contents of `output.txt`. Use `bash` shell command `diff` to compare your implementation's output with the correct output, like this:

```
diff -y --strip-trailing-cr --suppress-common-lines your-output.txt
output.txt
```

Options `-y`, `--strip-trailing-cr`, and `--suppress-common-lines` are described here. If `diff` completes the comparison without generating any output, then the contents of the two files are an exact match - that is, your source file generates the exact required output. If `diff` reports differences, you must go back and amend `q.c` to remove the reported differences.

## Submission and automatic evaluation

1. In the course web page, click on the submission page to submit `q.c`.
2. Read the following rubrics to maximize your grade. Your submission will receive:
    1. $F$ grade if your `q.c` doesn't compile with the full suite of `gcc` options.
    2. $F$ grade if your `q.c` doesn't link to create an executable.
    3. $F$ grade if outputs of functions `draw_tree` and `draw_animal` don't match correct outputs in `output.txt`.
    4. $C$ grade if *only one* of the outputs of functions `draw_tree` and `draw_animal` match correct outputs.
    5. $A+$ grade if outputs of functions `draw_tree` and `draw_animal` match correct outputs.
    6. A deduction of one letter grade for each missing documentation block. Your submission `q.c` must have one file-level documentation block and two function-level documentation blocks (one for function `draw_tree` and a second for function `draw_animal`). A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and the three documentation blocks are missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a $C$ grade and the three documentation blocks are missing, your grade will be later reduced from $C$ to $F$.