CAB202 Tutorial 4
Functions

Functions are the building blocks of complete programs. Much like how you wouldn't build a house from the roof down, programs are built by first creating functions, and then making use of them in a working program. Functions are used to represent a repeatable process (or algorithm) that adapts its behaviour based on the input arguments provided, and returns a result once completed. In the last session, you learnt about using conditional statements and loops to control the execution of code at runtime. By the completion of this tutorial, you will be able to use functions to represent repeatable segments of code, and know how to break complex problems down into smaller functions.

Defining a Function

The general form of a function definition in C programming language is as follows −

```
return_type function_name( parameter list ) {
   // body of the function
}
```

A function definition in C programming consists of a function header and a function body. Here are all the parts of a function −

- **Return Type** − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword `void`.
- **Function Name** − This is the actual name of the function, used to call the function within the users code base.
- **Parameters** − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** − The function body contains a collection of statements that define what the function does.

'Implicit Declaration' compiler warnings

The following code is a program that implements a function called `show_error_message()` which is implemented to print an error message (using the `printf()` function) based on an integer parameter (which indicates the type of error to print):

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

#define ERROR_EXAMPLE 1

// Declare a global boolean to check for an error
bool is_error = false;

void main(void) {

  while(1) {
      // We'll loop indefinitely
      // This is where some program logic would go
      if (is_error) {
        // If there is an error, run the error function
        show_error_message(ERROR_EXAMPLE);
      }
  }

}

void show_error_message(int error_number) {
  if (error_number == 1) {
    printf("ERROR (%d): Example error message\n", error_number);
  }
}
```

If you try to compile this program you will receive this compiler output:

```
example.c:20:6: warning: conflicting types for ‘show_error_message’ [enabled by default]
 void show_error_message(int error_number) {
      ^
example.c:14:9: note: previous implicit declaration of ‘show_error_message’ was here
         show_error_message(ERROR_EXAMPLE);
         ^
```

The AMS, by default, treats all compiler warnings as errors – which means this program would fail to compile. Why? The structure of the code is such that the `main()` function comes before the definition of the `show_error_message()` function – because of this, the compiler is not sure what the `show_error_message()` function is. The order of the code is very important in this case, since the compiler needs to know what the `show_error_message()` function does before it can execute this function. We can get rid of these warnings in a couple of ways:

- **Explicit ordering of function declarations** – While programming, we can make sure to order our function declarations from top to bottom, where the top function is the first to be called by our main function.
- **Include function Prototypes (aka function declarations)** – This is the most common method for preventing ‘implicit declaration’ warnings. A function declaration tells the compiler the name, return type, and argument types of a function – but does not include the code that actually runs when calling the function.

In order to remove the compiler warnings from our example program, we will add a function declaration at the top, so that the new program looks like:

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

#define ERROR_EXAMPLE 1

// Declare a global boolean to check for an error
bool is_error = false;

// Declare our show_error_message function
void show_error_message(int error_number);

void main(void) {

  while(1) {
      // We'll loop indefinitely
      // This is where some program logic would go
      if (is_error) {
        // If there is an error, run the error function
        show_error_message(ERROR_EXAMPLE);
      }
  }

}

void show_error_message(int error_number) {
  if (error_number == 1) {
    printf("ERROR (%d): Example error message\n", error_number);
  }
}
```

Non-Assessable Exercises

NOTE: These exercises are not assessable, and the tutor can help you with them. Assessable exercises are in the following section.

1. Draw the starting position of 5 racers in a game

The template provided in **`question_1.c`** on Blackboard is for a racing game where 5 players start on the left side of the screen and have to race to the right side of the screen. Before the race can begin, each player must be drawn in the leftmost column of the screen. Each player's character symbol is their player number. Players 1-5 must start in rows **4**, **8**, **12**, **16**, and **20** respectively

Write a function that draws a player at their start position with the following signature:

```
void draw_racer_at_start(int player_number);
```

If the variable **`player_number`** supplied is anything other than an **`int`** in the range 1 to 5, nothing is drawn. Only drawing should happen in this function (i.e. `show_screen()` should **not** be called)!

Challenge exercises:

- *Replace your solution with an analytical version (you should have nothing more than an* **`if`** *statement, with only one case, and a* ***single*** *draw call)!*
- *Extend your solution to work on any screen size (**Hint:** convert the values specified above to ratios of the screen height).*

2. Function Prototypes

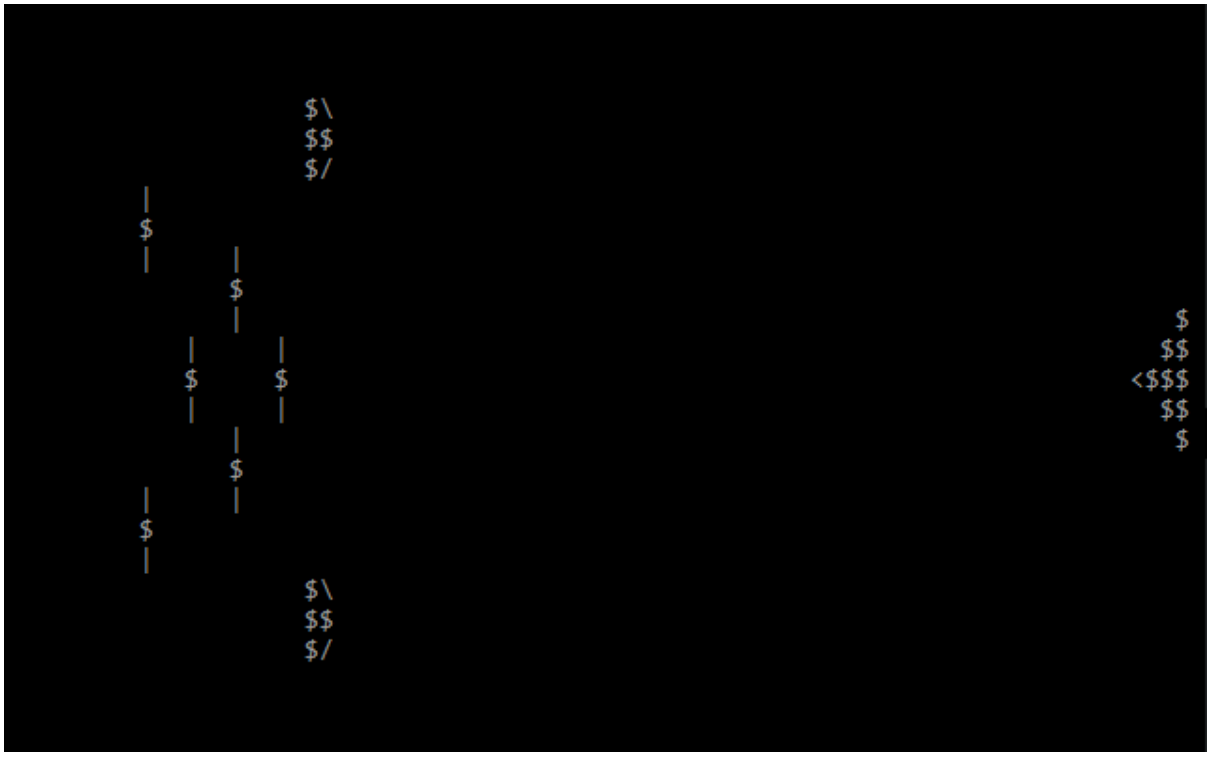
The program contained in **question_2.c** on Blackboard is a program containing many implicit declaration warnings. Your task is to find a way to remove these warnings. You may use either method from this tutorial to remove the warning – however, one method will be significantly easier.

Challenge exercises:

- *Refer to the ZDK include files and adapt your solution to use included C header files (*`.h files`*)*
- *Look up the purpose of function prototypes when compiling programs of multiple* **`.c`** *files*
- *How could you use header/source code separation in your assignment?*

3. Create 'space invaders' functions to work on top of 'cab202_graphics'

Imagine you are creating a space invaders game. The game will involve drawing the player's ship, and different types of enemies numerous times throughout the duration of a single game. This is a scenario where the use of functions, and possibly creating a library, are crucial programming decisions. Complete the following exercises in order, to slowly build up a more complete and reusable solution to draw the graphics for a space invaders game (the shape of the each of the ships is shown in the image below):



1. Create a **`question_3.c`** source file. Declare and implement the following 3 functions (the **x** and **y** arguments should correspond to the position of each ship's 'nose'):

```
draw_enemy_1(int x, int y);
draw_enemy_2(int x, int y);
draw_player(int x, int y);
```

2. Implement a `main` function that draws 6 type 1 enemies, 2 type 2 enemies, and 1 player (like the figure above). Compile and run your code to verify everything works properly.

Challenge exercises:

- *Combine your code with the solution from question 3. When the space bar is pressed, a bullet should shoot from the player's ship. Implement basic collision detection with the enemy ships directly in the path of the bullet.*
- *Create an **`invaders`** library (move your function declarations and implementations from above into **`invaders.h`** and **`invaders.c`** respectively). Your **`game.c`** file should now only have the `main` function. Add the necessary `#include` directive.*
- *Write a `Makefile`, or use a `gcc` compile command, to create **`libinvaders.a`** (look at the `Makefile` from the **ZDK**, or **`dummy_library`** for ideas).*

Assessed Exercises (AMS)

Complete the assessed exercises via the AMS (available at http://bio.mquter.qut.edu.au/CAB202).