Procedures & Functions

- All executable code resides within a **function**
- So far, the only function we have written is called **main**, which served as the entry point for our programs.
- A **function** is a named block of code that performs a task and then returns control to a caller.
  - The **caller** is the function that **invoked** the function
  - The **callee** is the function being **invoked**
  - You can think of the caller as the "parent" to the callee
- Because a function is just a block of code, we can call it multiple times throughout a program's execution
- After finishing, the function will branch back (return) to the caller.
- Consider this trivial example:
  - Suppose you want to print out the first 5 squares of numbers, do some processing, then print out the first 5 squares again. So far, we may write something like:

```c
#include <stdio.h>

int main(void)
{
  int i;
  for(i=1; i <= 5; i++)
  {
     printf("%d ", i*i);
  }
  // amazing things
  for(i=1; i <= 5; i++)
  {
     printf("%d ", i*i);
  }
  return 0;
}
```

- We wrote the same loop twice!
  - This is bad.
  - If we want to modify this code, to say print the first 5 cubes of numbers, we'd have to change code in two places
  - If we write a function to print the first 5 squares of numbers and call that function twice:

```c
#include <stdio.h>

```

```c
void print_squares(void)
{
  int i;
  for(i=1; i <=5; i++)
  {
    printf("%d ", i*i);
  }
}

int main(void)
{
  print_squares();
  // amazing things
  print_squares();
  return 0;
}
```

Functions

- Functions operate as *black boxes*, meaning they take input (parameters/arguments), do something with the input (function body), and spit out the answer (return value)
  - A function may not require any input at all (like our example above) and it may not return anything (like our example above - printing is not a form of returning).
- Terminology:
  - A function *f* that uses another function *g* is said to *call g* (i.e. *f* is the caller of *g*). * A function's inputs are known as its arguments (or parameters).
  - A function *g* that gives some kind of data back to the caller *f* is said to return that data.
- Let's look at a function to square the input of an integer:

```c
// the first int indicates that this function will return an integer to
    the caller
// square is the name of the function
// everything inside of the () are the function's arguments
// int x specifies a single argument named x of type int
int square(int x)
{
  // function body start
  int square_of_x;
  square_of_x = x * x;
  // return indicates what variable's value we return to the caller
  return square_of_x;
}
```

- A much simpler implementation:

```c
int square(int x)
{
    return x * x;
}
```

Function Syntax

- Functions take the form:

```c
type name(type1 arg1, type2 arg2, ...)
{
  /* function body code */
}
```

- *type* is the return type of the function
  - Could be **`int`**, **`float`**, etc
  - Can be **`void`** to indicate no return value
    * When a function is **`void`** type, you do not place a **`return`** in the function body
    * Example void function

```c
void print_hello(int number_of_times)
{
  int i;
  for(i=1; i <= number_of_times; i++) {
    printf("Hello!\n");
  }
}
```

- What about a function that takes no arguments?

```c
float calculate_number() // or you can explicitly place void as the
    argument -> e.g. (void)
{
  float result=1;
  int i;
  for(i=0; i < 100; i++) {
    result += 1;
    result = 1/result;
  }
  return result;
}
```

Function declarations

- A *function declaration* tells the compiler about a function's name, return type, and parameters.
- So far, we have looked at *function definitions*, which provide the actual code a function will execute.
- We can declare a function without defining it (similar to declaring a variable without initializing it)
- Function declarations take the following form:

```
type function_name(type1 arg1, type2 arg2);
```

- Notice the semicolon at the end - this is a statement in C
- Why bother with this?
  - Function declarations typically exist in header files (.h), and their corresponding definitions exist in a .c file of the same name
  - For instance, we have been writing `#include <stdio.h>`, which includes the stdio header
  - Many times people put `main()` at the top of their program, so a fellow programmer can see the program's entry point first
    * But a compiler reads a program top-to-bottom, so if you reference a function before the compiler is aware of its existence, the compiler won't know what to do (we'll see an example of this in a second)
  - But even for more complex programs, it's nice to see all of the functions in one area without having to scroll through every definition. Provides an overview of the functions available.
- Look back at our `print_squares` example. `print_squares` is before `main`. Let's try to move it after `main`:

```
#include <stdio.h>

int main(void)
{
  print_squares();
  // amazing things
  print_squares();
  return 0;
```

```c
 9 }
10
11 void print_squares(void)
12 {
13   int i;
14   for(i=1; i <=5; i++)
15   {
16     printf("%d ", i*i);
17   }
18 }
```

- We'll get a compiler error when we try to build this. Why?
  - When the compiler reads line 5, it has no idea what `print_squares` is. Function declarations let us tell the compiler: "This function will be defined later. When the linker executes (third stage of compilation), this function will be defined, compiled, and ready for linking"
- Let's add a function declaration to fix:

```c
#include <stdio.h>

void print_squares(void);

int main(void)
{
  print_squares();
  // amazing things
  print_squares();
  return 0;
}

void print_squares(void)
{
  int i;
  for(i=1; i <=5; i++)
  {
    printf("%d ", i*i);
  }
}
```

- It builds! The compiler is aware that `print_squares` is a function and will be defined later.

Static functions

- Static functions can only be called from the file in which they were written
- This helps protect functionality from being available in other files. Essentially makes the function private to this particular file
- Example:

```c
static int less_than( int a, int b )
{
    return (a < b)? a : b;
}
```

Calling functions

- Say we wanted to call the `calculate_number` function.
- Remember this function takes no arguments and returns a float
- We would write:

```c
float f;
f = calculate_number();
```

- If you do not assign the return value to a variable, the return value is discarded (will not error).
- What if the function takes arguments?

```c
int square_of_10;
square_of_10 = square(10);
```

- We can also pass appropriately (correctly) typed variables instead of literals

```c
int square_of_x;
int x = 10;
square_of_x = square(x);
```

- C will attempt to type cast whatever you pass into the appropriate type.
  - For instance, if you pass a floating point number for an int argument, the floating point number will be type cast into an int
- If the function doesn't return anything, simply call the function

```c
print_hello();
```

Functions from the C Standard Library

- Wide range of functions already written for you!
- No need to reinvent the wheel
- These exist to make your life easier
- https://en.wikibooks.org/wiki/C_Programming/Procedures_and_functions#Functions_from_the_C_Standard_Library

Variable-length Argument Lists

- Functions don't have to specify *exactly* how many arguments they take
- For instance, imagine you wanted to write a function to compute the average of a set of numbers.
  - So far, we have no way of handling *N* numbers.
  - We could write a function for averaging 2 numbers, 3 numbers, etc, but that would be very painstaking.
- Where have we seen functions that take in an arbitrary number of arguments already?
  - `printf`
  - `scanf`
- In order to write a function that takes a variable number of arguments, first include the `stdarg.h` header

Steps:

1. Declare the function as you normally would
2. Last argument to the function is an ellipsis `...` to indicate there is a variable list of arguments

- Example function declaration:

```c
float average (int n_args, ...);
```

- Somehow we need to specify how many arguments are in the list.
- Above we did this with the `n_args` argument
- Next, we need a mechanism to access the list of arguments. We'll declare a variable for the list of arguments:

```c
va_list myList;
```

- Notice the type here, `va_list`. This type is provided by `stdargs.h`

- To actually use myList, we must assign it a value. The va_start macro (similar to a function for now)
  - The va_start macro takes two arguments:
    1. The va_list you plan on storing values in
    2. The name of the last variable appearing before the ellipsis

```c
#include <stdarg.h>
float average (int n_args, ...)
{
  va_list myList;
  va_start (myList, n_args);
  va_end (myList);
}
```

- Now we have done all of the setup required to use the list
- To actually access a value in this list, we use the va_arg macro, which "pops" off the next argument in the list
  - In the va_arg marco, you provide:
    1. The va_list variable to pop the value from (e.g. myList)
    2. The type of the variable being extracted

```c
#include <stdarg.h>
float average (int n_args, ...)
{
  va_list myList;
  va_start (myList, n_args);

  int myNumber = va_arg (myList, int);
  va_end (myList);
}
```

- By popping n_args integers off the variable-length argument list, we can find the average of all numbers:

```c
#include <stdarg.h>

float average (int n_args, ...)
{
    va_list myList;
    va_start (myList, n_args);

    int numbersAdded = 0;
    int sum = 0;

    while (numbersAdded < n_args) {
        int number = va_arg (myList, int); // Get next number from list
        sum += number;
        numbersAdded += 1;
    }
    va_end (myList);

    float avg = (float)(sum) / (float)(numbersAdded); // Find the average
    return avg;
}

int main(){
  float avg = average(2, 10, 20);
}
```

- If we call this function with 2, 10, and 30, we get the average of 10 and 20, which is 25:
  - `average(2, 10, 20);`

Exercises

1. What is the effect of calling show(4)?

```c
int show(int x) {
  printf("%d %d\n", x, x*x);
  return x*x;
  printf("%d %d\n", x, x*x*x);
  return x*x*x;
}
```

2. What does the following C function do?

```c
int eq3(int a, int b, int c) {
  if ((a == b) && (a == c))
    return 1;
  else
    return 0;
}
```

3. Write a C function that takes a real number as an argument and returns the absolute value of that number.

```c
#include <stdio.h>

float absolute(float n){
  if (n < 0.0){
    return -n;
  } else{
    return n;
  }
}

int main(){
  float abs1 = absolute(5.5);
  float abs2 = absolute(-10.2);
  printf("5.5 is %f, -10.2 is %f\n", abs1, abs2);
}
```

4. Write a C function that takes in N integers as arguments and returns the value of the largest one.

```c
#include <stdio.h>
#include <limits.h>
#include <stdarg.h>

int find_largest(int n_args, ...){
  va_list numbers;
  va_start (numbers, n_args);

  int largest = -2500000; // should use INT_MIN from limits.h instead
  int num_processed = 0;
  while (num_processed < n_args){
    int number = va_arg(numbers, int);
    if (number > largest){
      largest = number;
    }
    num_processed++;
  }
  va_end(numbers);
  return largest;
}

int main(){
  int largest = find_largest(4, -10, 30, 40, 50);
  printf("The largest number is %d\n", largest);
}
```

5. Write a C function to calculate a total cost of a meal. The function should take in a base cost, the tip percentage as a decimal, and a tax percentage as a decimal. The function should return the total cost of the meal.

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

float tip_calculator(float base, float tip_pct, float tax_pct){
  float total = base + base * tip_pct + base * tax_pct;
  return total;
}

int main(){
  float total = tip_calculator(35.6, 0.2, 0.01);
  total = roundf(total * 100) / 100;
  printf("Total is %.2f\n", total);
}
```