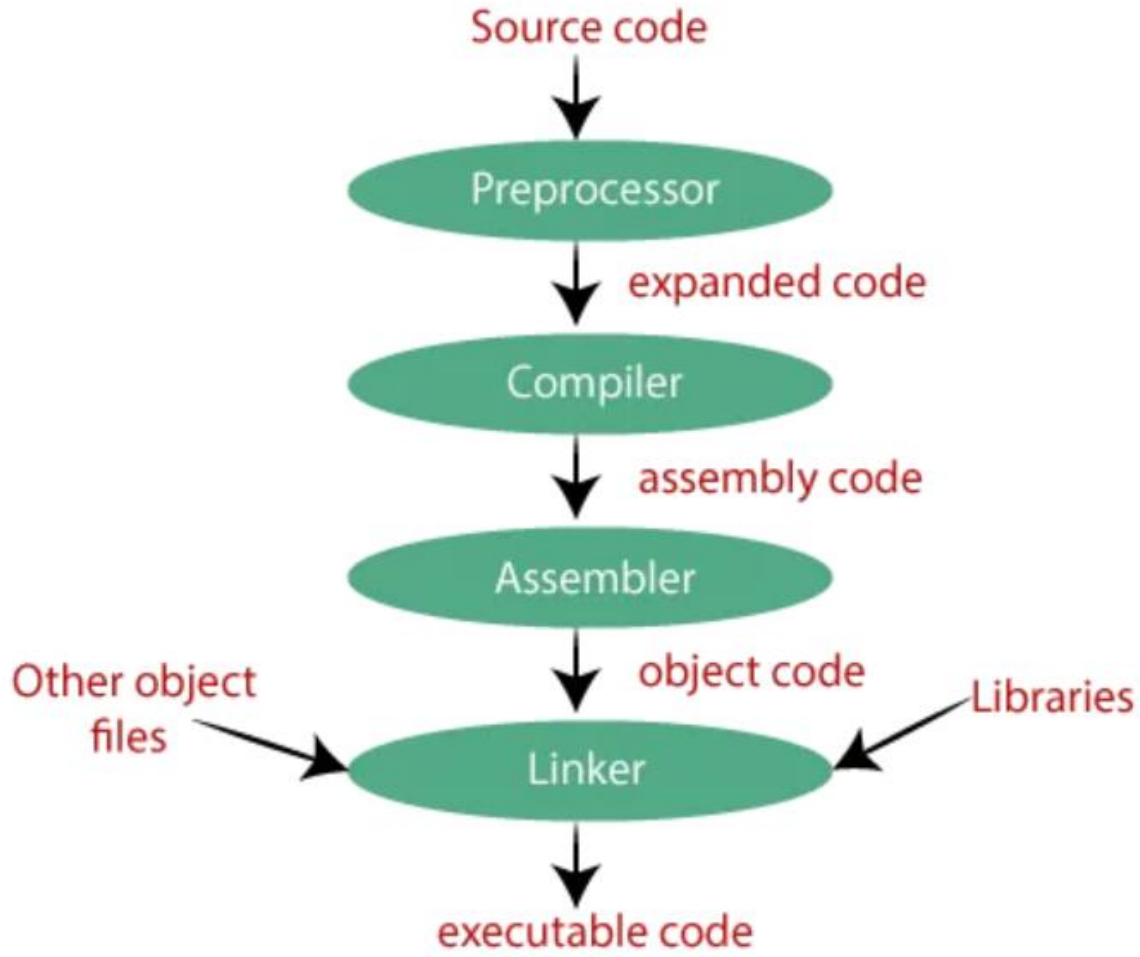
The C Language

Required tools

- We need some tools to compile a C program, these are:
    - a preprocessor
    - a compiler
    - an assembler
    - a linker



Compilation steps and running

- Save the simple C program into a file called ***myprog.c***
- Execute the following command in the shell to compile
  - *gcc myprog.c*
- The command above creates a file named ***a.out***; to run it, execute
  - ./a.out

A simple C program

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

The structure of a source file: headers inclusion

- At the beginning of a source file people tend to include the header files in which the functions signatures are declared, e.g.

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

- E.g.: ***printf*** functions is declared in ***stdio.h*** file

The structure of a source file: *main* function

- An executable file must include a ***main*** function definitions

```
int main() {
            .
            .
            .
}
```

- The execution of a program begins from the ***main*** function

Variables

What is a variable?

- A variable is named object defined by us that has a ***value***, a ***data type***, and a ***memory address***.
- E.g.: *mychar* and *num* are variables.

```
char mychar = -15;
int num = 99;
```

A pictorial view of a variable

- You can regard a variable as a box that contains a value and has an address in the memory.

Examples

```
int main() {
    char mychar = -15;
    int num = 99;
    double PI = 3.14;
        .
        .
        .
}
```

Data Types

Data Types

- C is a statically typed language; that is, you have to specify a data type when you declare/define variables and functions (Unlike Python, Like Java).
- Every data type occupies a size of memory, and support only limited range of values; so, before choosing a data type for your variable or functions, take these into consideration.

Data Types

Type	Size(in 64bit machine)	Range
char	1 byte	[-128 to 127] or [0 to 255] (*implementation specific)
signed char	1 byte	[-128 to 127]
unsigned char	1 byte	[0 to 255]
short	2 bytes	[-32768 to 32767]
unsigned short	2 bytes	[0 to 65535]
int	4 bytes	[-2,147,483,648 to 2,147,483,647]
unsigned int	4 bytes	[0 to 4,294,967,295]
long	8 bytes	[-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807]
unsigned long	8 bytes	[0 to 18,446,744,073,709,551,615]
float	4 bytes	3.4E +/- 38 (7 digits)
double	8 bytes	1.7E +/- 308 (15 digits)

Range calculation for integers

- If we have ***n*** bit spaces, then the range of an integer type is calculated as follows
  - For signed : $[-2^{n-1}, 2^{n-1} - 1]$
  - For unsigned: $[0, 2^{n} - 1]$
- E.g.
  - For 1 byte ***char*** type
    - For signed: $[-2^{7}, 2^{7} - 1] = [-128, +127]$
    - For unsigned: $[0, 2^{8} - 1] = [0, 255]$
  - For 2 bytes ***short***
    - For signed: $[-2^{15}, 2^{15} - 1] = [-32{,}768, 32{,}767]$
    - For unsigned: $[0, 2^{16} - 1] = [0, 65{,}535]$

Simple I/O Operations

Including the header file

- To print something to the screen or read something from the keyboard we can use some of the functions defined in *Standard C Library.*
- The header file that contains I/O functions is **stdio.h**
- To use the functions declared in this header file, we must include it in our source file:

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

Printing

- To print something to the screen we could use ***printf*** function

```c
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    printf("My name is Ege.\n");
    printf("%d\n", 15);
    printf("%d\n", 15 * 21);
    printf("PI: %f", 3.14);
    return 0;
}
```

```c
#include <stdio.h>

int main() {
    float PI = 3.14;
    int radius = 5;
    float area = PI * radius * radius;
    printf("Area: %f", area);
    return 0;
}
```

printf

- ***printf*** is a function that is used for formatted printing
- Declaration: `int printf(const char *format, ...)`
- We can pass these values as `*format`
  - Only string: `"This is a string"`
  - Only format specifiers: `"%d", "%f", "%c"`
  - Or both: `"The value is: %d"`
- We can print values to the screen the same number of times with format specifiers that we provide in the `*format` string

printf: format specifiers

- If you want to print a value to the screen you have to specify the format specifier in the format string as a placeholder
- Here are some:
  - %c: for characters
  - %d: for integers
  - %f: for real numbers
  - %s: for strings

Reading

- To read something from the keyboard we could use ***scanf*** function.

```c
#include <stdio.h>

int main() {
    int radius;
    printf("Radius: ");

    scanf("%d", &radius);

    printf("Area: %f", 3.14 * radius * radius);
    return 0;
}
```

scanf

- ***scanf*** is a function that is used for reading
- Declaration: `int scanf(const char *format, ...)`
- You should pass the corresponding format specifier in `*format` for your variables. If you want to read
  - an integer value, use `"%d"`
  - a real value use, `"%f"`
  - a character value, use `"%c"`
  - a string, use `"%s“`

scanf: e.g.

```
int x; scanf("%d", &x);
char y; scanf("%c", &y);
float z; scanf("%f", &z);
```

&: address-of operator

- ***scanf*** reads from keyboard into a variable. To do so, it needs the address of the variable, not its name. To pass the address of a variable into a function like ***scanf***, we use & (ampersand) operator. E.g.
  - &my_var: indicates the address of *my_var*
    - `int my_var; scanf("%d", &my_var);`
- If we know the address in advance, we can also pass it. E.g.
  - `scanf("%d", 0xFFFFCC3C);`

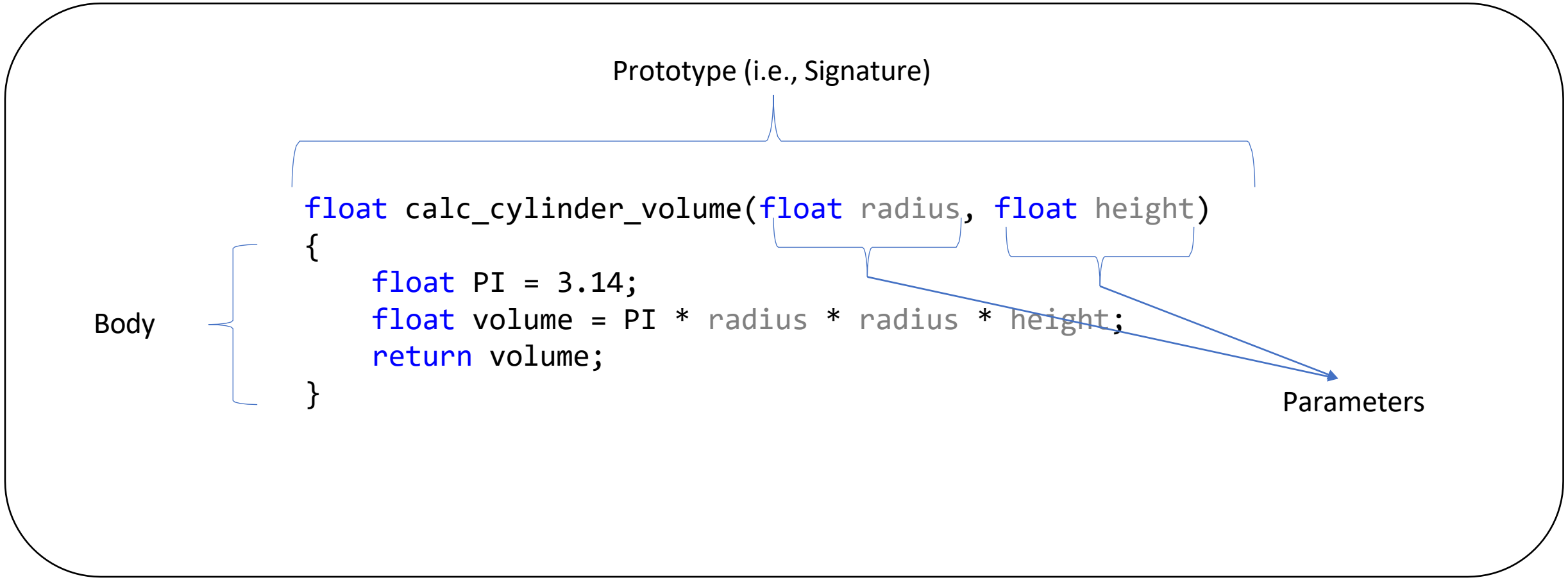
Functions

What is a function?

- A function is a group of statement that performs a certain task.
- Functions in C have memory addresses and introduce local scopes for variables defined in them. A function definition looks like this:

```
<return type> <function name> (<parameter1 type> <parameter1 name>,
                               <parameter2 type> <parameter2 name>,
                               …)
{


}
```

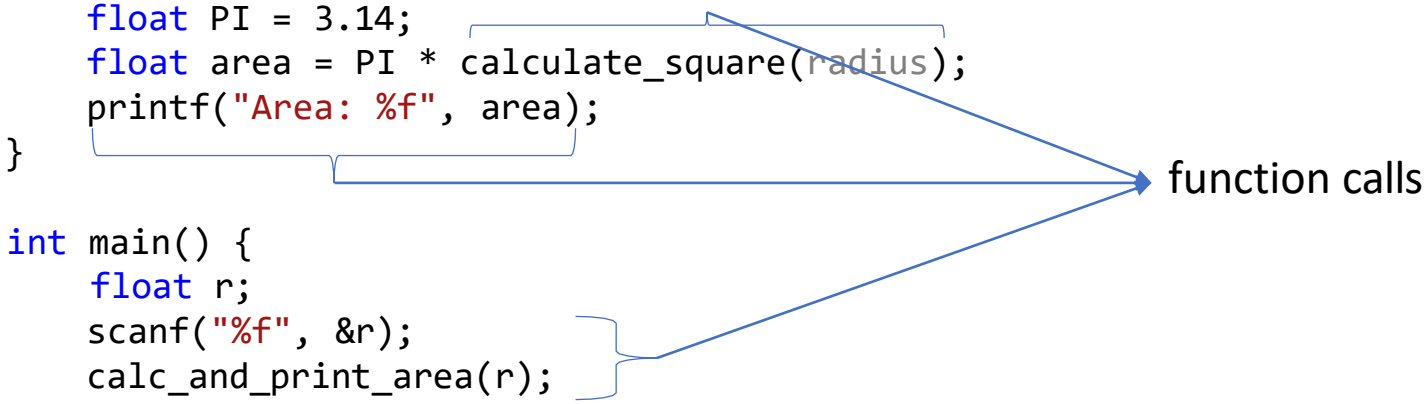
Terminology



This whole part is called “function definition”

E.g.

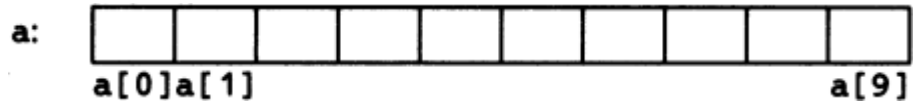
```c
#include <stdio.h>

float calculate_square(float val) {
    return val * val;
}

void calc_and_print_area(float radius) {
    float PI = 3.14;
    float area = PI * calculate_square(radius);
    printf("Area: %f", area);
}

int main() {
    float r;
    scanf("%f", &r);
    calc_and_print_area(r);
    return 0;
}
```



Arrays

What is an array?

- An array holds a set of variable that share the same data type in a contagious memory region



- E.g.: Here below is an integer array
  - ***my_array***: 95 21 -11 9 10 1005
- In C, the first element resides in 0^{th} index
  - 0^{th} element of my_array is 95
  - 1^{st} element of my_array is 21
  - 2^{nd} element of my_array is -11
  - …

Credits: The C Programming Language, Second Edition – K&R

Defining arrays

- The template for arrays stored in stack area is
  - <data type> <array name>[<number of elements>];
  - E.g.:

```
int my_array[15];
float the_holy_arr[1000];
```

- We can also initialize arrays with predefined values
  - E.g.:

```
int my_array[] = {1, 6, 9, -5, 28};
```

- Note: If you don't provide initial values for arrays defined in local scope, they contain junk values; that is, the values can be anything in the data type domain

Accessing array elements

- We can access any element of an array by providing an index. E.g.
  - We can access the 15th element of array ***my_array*** by my_array[14]
  - The 1st element by my_array[0]
  - The nth element by my_array[n-1]
- We consider arrays' elements as variables, by doing so we can apply all operators to them as we do on variables

```
my_array[9] = 3;
my_array[2] = my_array[1] + my_array[0];
if (my_array[3] > 15) {...}
int n = 2;
my_array[n+1] = 0;
```

Exercise

- Implement the following program in C. (Red texts indicate example user inputs.)

```
Enter the first positive integer: 3
Enter the second positive integer: 4

3 + 4 = 7 can be visualized as the following:

* * * + * * * * = * * * * * * *
```

- You can also implement an invalid input prompt for negative or zero inputs.

The END

Next week:

Pointers and Structs