FUNCTIONS

COMP1511 Week 4

Joanna Lin

What we've learnt so far

Understanding why...

- Variables allow us to store information and change the information we store.
- `If` statements allow us to run code only when some condition is satisfied.
- `while` loops allow us to run code repeatedly until some condition is not satisfied.
- these concepts are ultimately a product of trying to break logic down into very fundamental components and thinking about what the essential things are for programmers to communicate their thoughts to the computer.
- the nuances and features we see in the C programming language came from careful thought and experimentation by experienced programmers
  - it's good because we don't need to reinvent the wheel.
  - it's bad because abstract computing concepts seem to fly out of a vacuum, and we need to somehow fumble with it until it clicks.

Functions

What purpose do they achieve?

- Allows us to separate logic into modular components and reuse code that either we or other programmers have written.
- We can break large problems into smaller problems and focus specifically on solving the smaller problems.
- Allows us to hide away complicated details, to make reading programs much easier for our minds to handle
  - **Analogy:** it is much easier to tell a human to make a peanut butter sandwich than it is to give them a long list of specific instructions on how to make one. The instructions are inherent, but there is a name attached to it.
- But how is this achieved?

Using Functions: First Look

Notice how

- **`get_larger`** and **`main`** have very similar syntax
- The line where **`get_larger`** is called and the **`scanf`** and **`printf`** lines are very similar

Why is this the case?

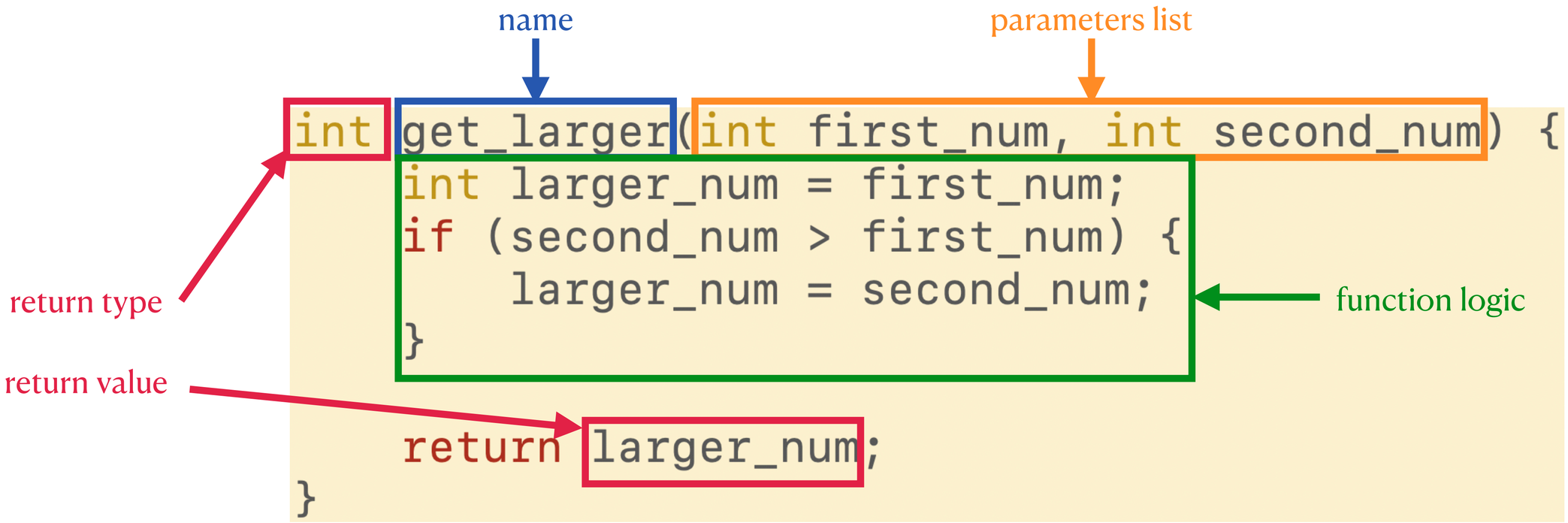
Function Definition

Function Call

```c
#include <stdio.h>

int get_larger(int first_num, int second_num) {
    int larger_num = first_num;
    if (second_num > first_num) {
        larger_num = second_num;
    }

    return larger_num;
}

int main(void) {
    int first;
    int second;
    scanf("%d%d", &first, &second);

    int larger = get_larger(first, second);
    printf("The larger number is %d\n", larger);
    return 0;
}
```

Using Functions: Definition

We need to define a function to be able use it



Using Functions: Calling

To use the function, we must call it somewhere:

```c
int main(void) {
    int first;
    int second;
    scanf("%d%d", &first, &second);

    int larger = get_larger(first, second);
    printf("The larger number is %d\n", larger);
    return 0;
}
```

(variable type we store the return value in must match the return type)

name

arguments (must match the types of the parameters list)

As with all variable assignments, the right side is evaluated first. Once the function returns, we can think of the function call as being replaced by whatever it returned.

Alternatively...

```
int main(void) {
    int first;
    int second;
    scanf("%d%d", &first, &second);

    printf("The larger number is %d\n", get_larger(first, second));
    return 0;
}
```

Issues?

- The **`main`** function contains code that tells us what the program actually does, so we want it to be as close to the top as possible in our file.
  - If we define a lot of functions, **`main`** will be pushed very far down.
- If we put **`get_larger`** after **`main`**, our code will not compile, because **`main`** wouldn't know that **`get_larger`** exists
- What is the solution?

```c
#include <stdio.h>

int get_larger(int first_num, int second_num) {
    int larger_num = first_num;
    if (second_num > first_num) {
        larger_num = second_num;
    }

    return larger_num;
}

int main(void) {
    int first;
    int second;
    scanf("%d%d", &first, &second);

    int larger = get_larger(first, second);
    printf("The larger number is %d\n", larger);
    return 0;
}
```

Using Functions: Prototype

We often want the function to be defined after the function that calls it.

To let **`main`** know that **`get_larger`** exists, we need to give **`main`** a prototype of the function **`get_larger`**.

This is just to let **`main`** know

- the parameter list so that arguments are of correct type, and
- what the return type is so that the return value can be stored in the correct variable type.

So actually, the variable names given in the prototype are redundant. An alternative (but not recommended) way of writing the prototype is

```
int get_larger(int, int);
```

```
#include <stdio.h>

int get_larger(int first_num, int second_num);

int main(void) {
    int first;
    int second;
    scanf("%d%d", &first, &second);

    int larger = get_larger(first, second);
    printf("The larger number is %d\n", larger);
    return 0;
}

int get_larger(int first_num, int second_num) {
    int larger_num = first_num;
    if (second_num > first_num) {
        larger_num = second_num;
    }

    return larger_num;
}
```

`void` Functions

Functions don't need to return anything

- Sometimes, we just want functions to just do stuff without giving any output back to the function that called it
- For example, we could've written a function **`print_larger`** that prints the larger of the two values passed into it.
- The return type is **`void`**

```c
#include <stdio.h>

void print_larger(int first_num, int second_num);

int main(void) {
    int first;
    int second;
    scanf("%d%d", &first, &second);
    print_larger(first, second);

    return 0;
}

void print_larger(int first_num, int second_num) {
    int larger_num = first_num;
    if (second_num > first_num) {
        larger_num = second_num;
    }
    printf("The larger number is %d\n", larger_num);
}
```

POP QUIZ!

Confusion Hotspots

Question 1

Variable Name Clashes

Notice how in the following code, the variable **`larger`** is declared in both **`main`** and **`get_larger`** function.

Is the code still valid?

Yes

Why/Why not?
Because functions have a ‘scope’ separate from each other. That means they cannot see the variables that have been declared in other functions. This is great because when we write functions, we can dedicate our focus to writing its logic without worrying about name clashes and screwing up other parts of the program.

```c
#include <stdio.h>

int get_larger(int first_num, int second_num);

int main(void) {
    int first;
    int second;
    scanf("%d%d", &first, &second);

    int larger = get_larger(first, second);
    printf("The larger number is %d\n", larger);
    return 0;
}

int get_larger(int first_num, int second_num) {
    int larger = first_num;
    if (second_num > first_num) {
        larger = second_num;
    }

    return larger;
}
```

Question 2

Changing the values of arguments

What is the value of `num` in `main` before and after the `change_number(num);` line?

15 and 15.

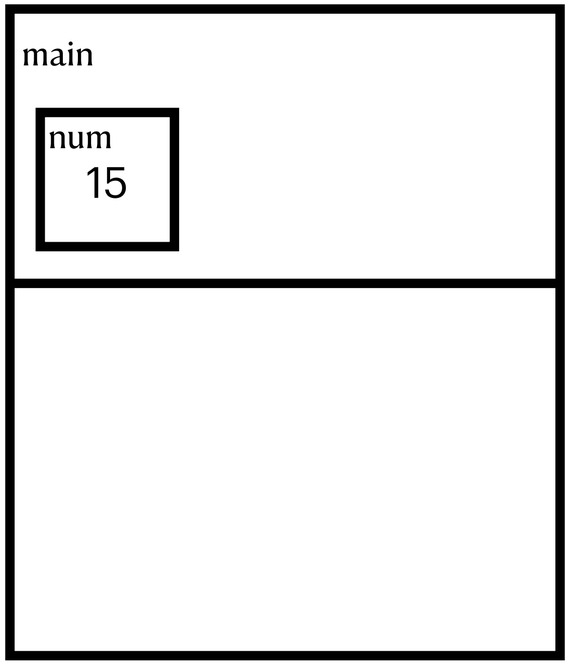
Why is this the case?

Arguments are passed in ‘by value’. The value of the arguments are copied into the block of memory given to the function being called.

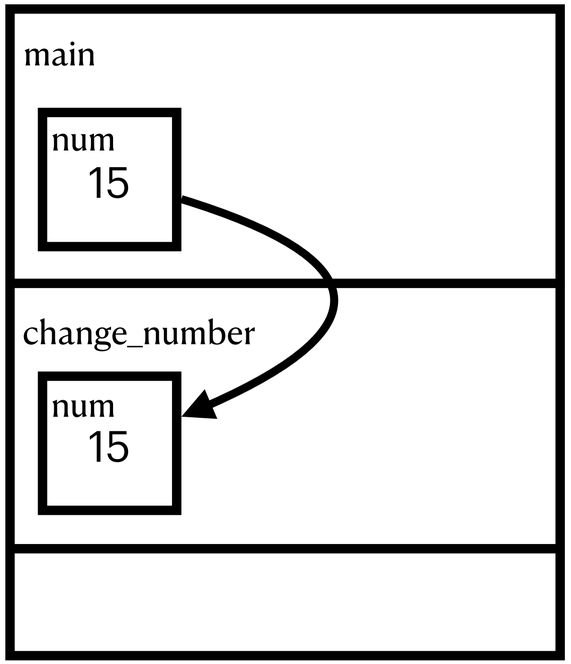
```c
#include <stdio.h>

void change_number(int num);

int main(void) {
    int num = 15;

    printf("Before: %d\n", num);
    change_number(num);
    printf("After: %d\n", num);

    return 0;
}

// Change the given variable "num" to be the value 10
void change_number(int num) {
    num = 10;
}
```

Memory Model: Function Calls

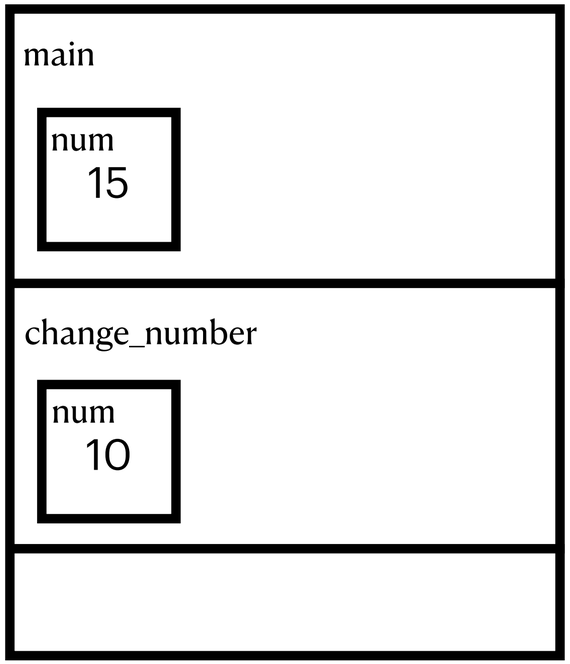
(Computer memory)



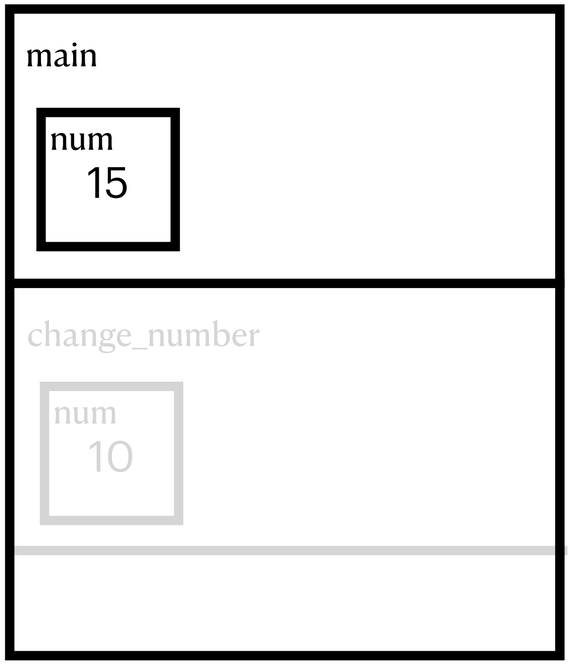
In the beginning, **`main`** has its own block of memory. The variable **`num`** is given the value 15.



`main` calls **`change_number`**. The value 15 is copied over to the block of memory for **`change_number`**



`change_number` changes the value of **`num`** at the block of memory that was assigned to it.



when we exit **`change_number`**, its memory is destroyed and the **`num`** in **`main`** remains unchanged

General Tips...

- If you're not feeling confident about functions, try typing out all of the code in these slides **from scratch** without looking, and make sure you understand what each line is doing.
    1. Write a function that takes in two arguments and returns the larger number
    2. Write a function that takes in two arguments and prints the larger number.
    3. Experiment! Play around with the prototype and arguments.
- Get further practice in typing out functions from scratch to get familiar with the syntax.
    - The functions shown in this week's lab exercises may not be super useful in helping you write neater looking programs, so it might be hard to see why you'd need them. However, they will get you more familiar with the syntax.
    - You'll definitely start to see the benefits of using functions when you start doing your assignment.
- Give time for the concept of functions to settle in. Functions will start to click more the more you force yourself to make use of it.