COMP1511 Week 4

Arrays and Functions

Joanna Lin

Reminders

- Labs are due next **Monday 8pm**, as usual.
- Assignment 1 has come out.

Functions

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 on solving the smaller problems.
- Allows us to abstract away complicated details from `main` to make understanding the main function easier
  - **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.

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?

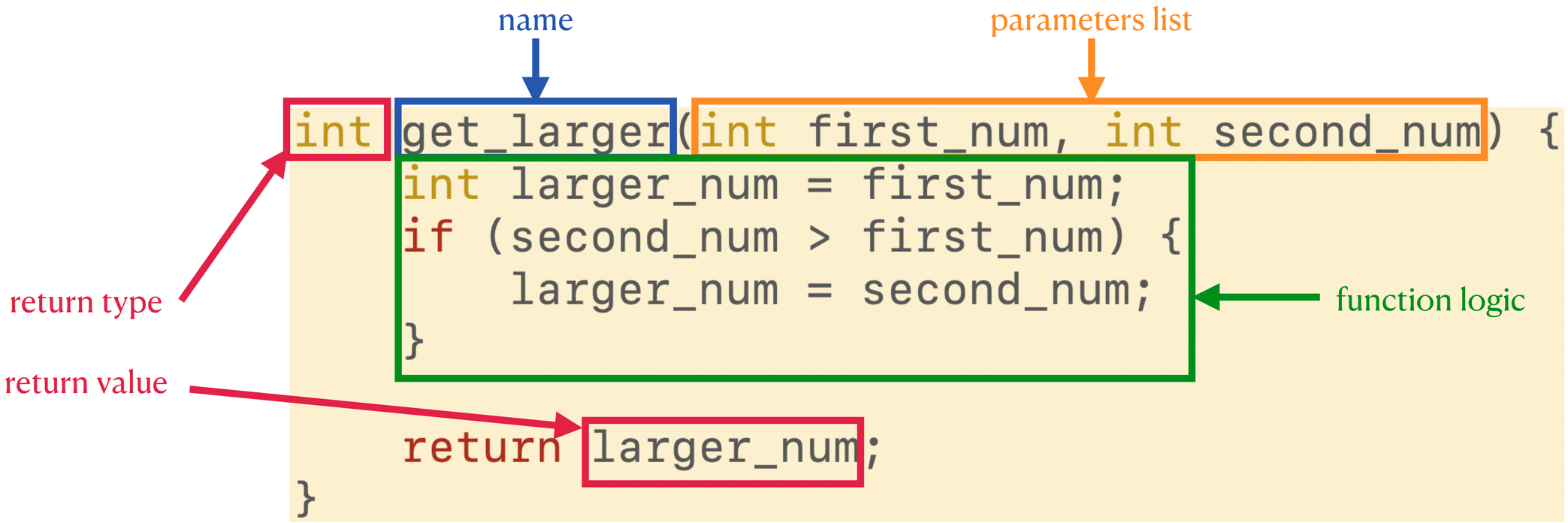
Function Definition

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...

We can replace a `%d` with the return value directly!

```
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

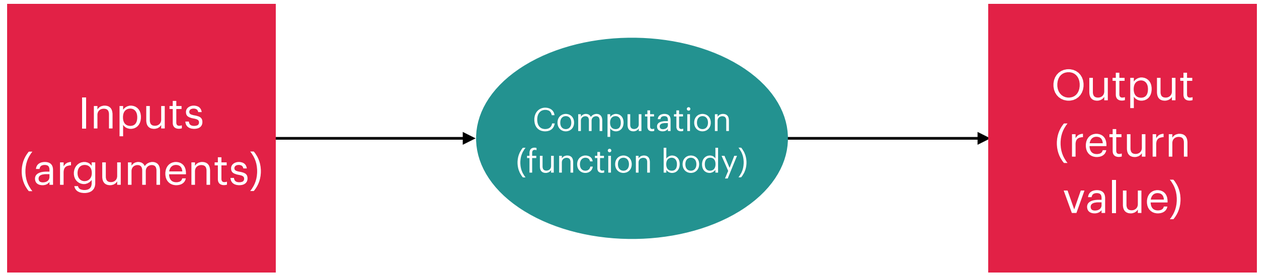
- the parameter list so that it can pass in arguments of the correct type, and
- what the return type is so that the return value can be stored in the correct variable type.

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);
```

```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_num = first_num;
    if (second_num > first_num) {
        larger_num = second_num;
    }

    return larger_num;
}
```

Simplified View of Functions



`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

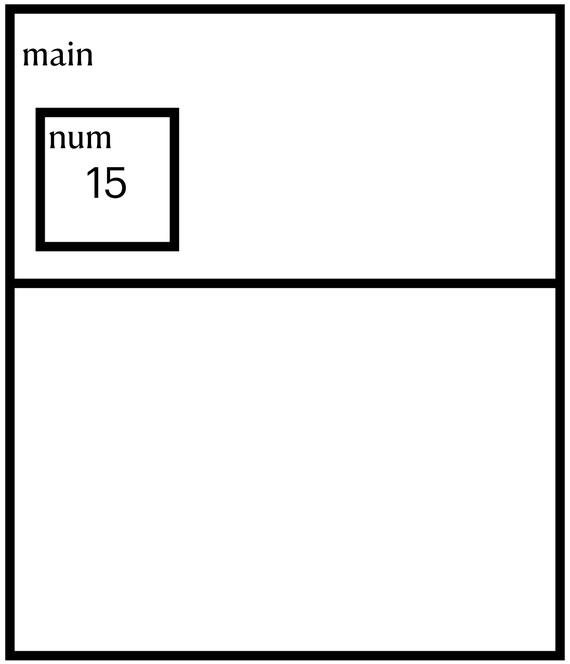
Why is this the case?

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

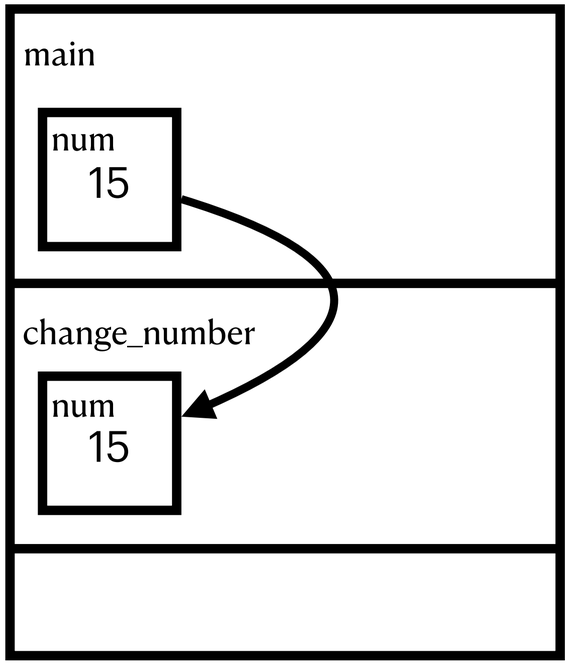
```c
#include <stdio.h>

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

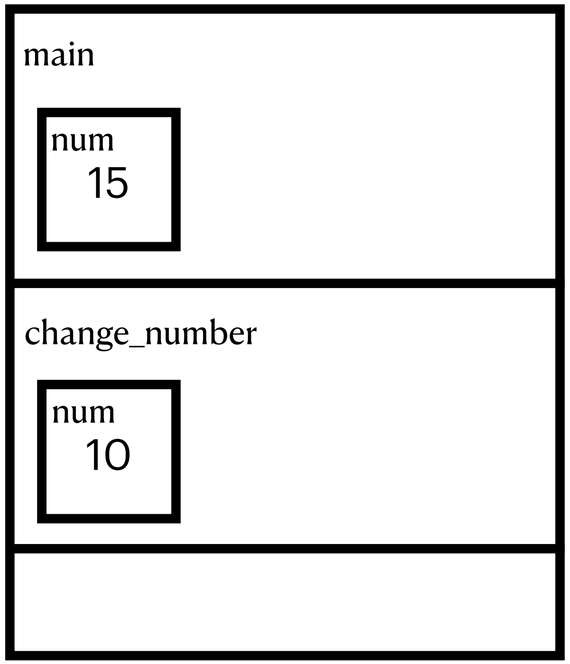
(Computer memory)



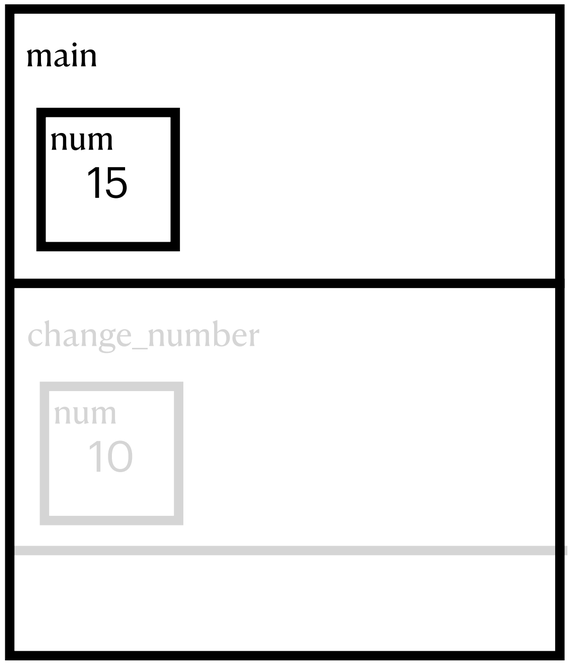
In the beginning, **`main`** has its own block of memory. The variable **`num`** is given the value 15.



`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

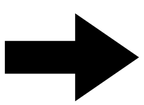
Arrays

Arrays

What they are and how to use them

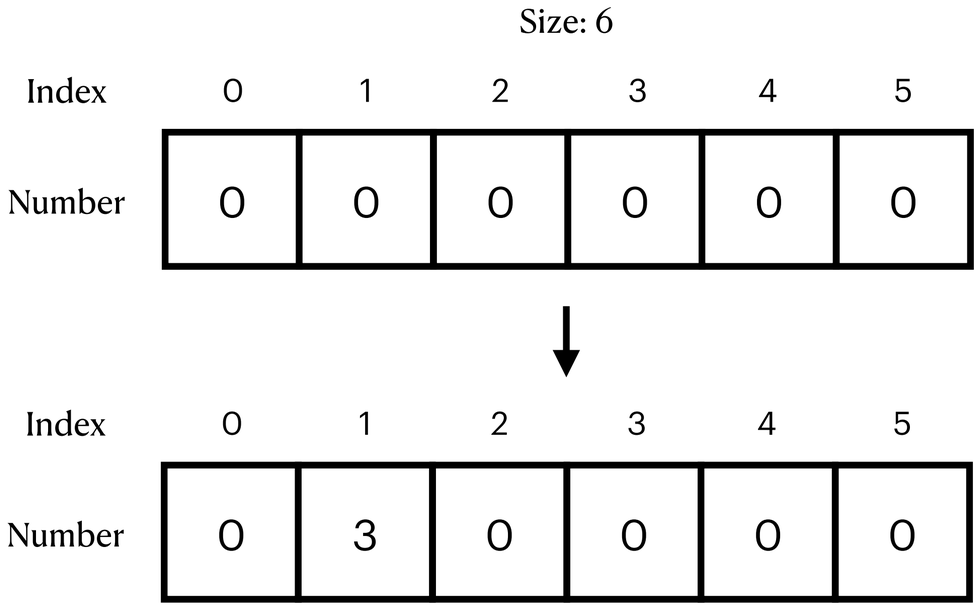
- A ‘container’ storing many variables of the same type in a contiguous block of memory.
  - We can essentially create many variables in one line.
- Declare with **`type array_name[SIZE]`**; (elements aren’t automatically initialised but we can initialise all elements to 0 using the code below)
- Each element can be accessed using an index, starting at 0.

```
int num1 = 0;
int num2 = 0;
...
```



```
int array[6] = {0};

array[1] = 3;
```

Note that the square bracket **[]** notation has two different meanings when used during and after declaration.
During declaration, the number indicates the **size** of the array.
After declaration, the number indicates the **index** of the element we want to access



Arrays

The Can't's

- The initialising to `0` trick does not work with any other number.

```
int array[6] = {1};
```

What really happens is...

0	1	2	3	4	5
1	0	0	0	0	0

- We cannot assign an array to another array.

```
int array1[6] = {0};
int array2[6] = array1;
```

Arrays

The Do's

- Declare array sizes with **`#define`**'d constants `#define SIZE 6`
- Use arrays with while loops. Be careful with deciding the condition on the loop!

printing the elements of a loop

```c
int array[SIZE] = {};
int i = 0;
while (i < SIZE) {
    printf("%d\n", array[i]);
    i++;
}
```

✅

notice how the condition on the loop is strictly less than the size of the array

spot the mistake!

```c
int array[SIZE] = {};
int i = 0;
while (i <= SIZE) {
    printf("%d\n", array[i]);
    i++;
}
```

⚠️

the valid indices of an array are `0` to `SIZE - 1` **NOT** `SIZE`

- Keep functions in the know of how large an array is if it needs to loop through the array. We can't retrieve the size with the array alone!

Changing Arrays With Functions: Will this work?

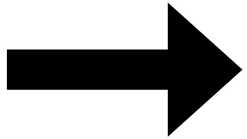
Answer: Yes... but why?

```c
#include <stdio.h>

#define SIZE 1000

void change_array_element(int arr[SIZE], int index);

int main(void) {
    int array[SIZE] = {0};

    int index;
    printf("Enter index to change: ");
    scanf("%d", &index);

    printf("Before function: %d\n", array[index]);
    change_array_element(array, index);
    printf("After function: %d\n", array[index]);
}

// Change an element of an array at a given index to
// now have 42
void change_array_element(int arr[SIZE], int index) {
    arr[index] = 42;
}
```

Using Arrays With Functions

Cleaning up our main function

```c
int main(void) {
    int square_nums[SIZE] = {0};

    // Writes square numbers to an array.
    int i = 0;
    while (i < SIZE) {
        square_nums[i] = (i + 1) * (i + 1);
        i++;
    }

    // Prints values in square_nums array.
    int j = 0;
    while (j < SIZE) {
        printf("%d\n", square_nums[j]);
        j++;
    }
    return 0;
}
```



```c
void write_square_nums(int size, int array[]);
void print_array(int size, int array[]);

int main(void) {
    int square_nums[SIZE] = {0};
    write_square_nums(SIZE, square_nums);
    print_array(SIZE, square_nums);
    return 0;
}

// Writes square numbers to an array.
void write_square_nums(int size, int array[]) {
    int i = 0;
    while (i < SIZE) {
        array[i] = (i + 1) * (i + 1);
        i++;
    }
}

// Prints elements in an array, each on a new line.
void print_array(int size, int array[]) {
    int i = 0;
    while (i < SIZE) {
        printf("%d\n", array[i]);
        i++;
    }
}
```

Notice how much shorter the main function is.

We also don't have to keep track of the loop variables we've used because there's only one loop in each function!

`scanf`'s in loops

An aside...

- Remember **`scanf`** returns the number of variables it successfully scanned in.
- This means we can use **`scanf`** in a while loop to collect user input until the user enters **`Ctrl + D`** (**`EOF`**), or any input that **`scanf`** doesn't expect.

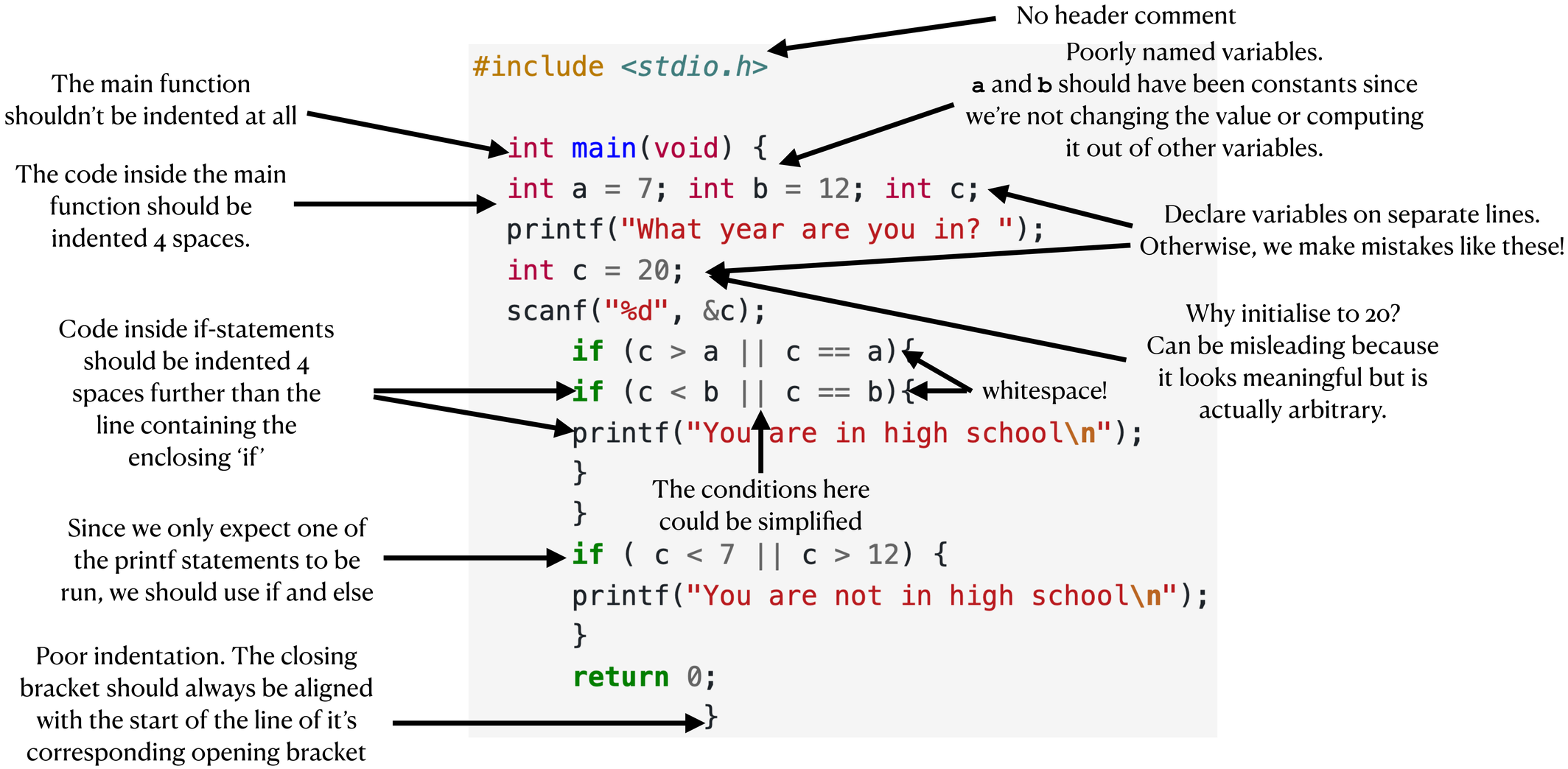
```c
// Program that scans in characters until EOF and prints
// them if they are letters
#include <stdio.h>

int is_letter(char c);

int main(void) {
    char c;
    while (scanf(" %c", &c) == 1) {
        if (is_letter(c)) {
            printf("%c\n", c);
        }
    }
}

int is_letter(char c) {
    return ('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z');
}
```

note: (needed for characters) to make **`scanf`** ignore newlines, remember we need a space before the **`%c`**

note: one-liner functions returning just a condition is an effective way to clean up your code!

Code Review 1

What's wrong with this?



Beautified 👸

```c
// Program that tells the user whether they're in high
// school or not
// Joanna Lin (z5311239), October 2021

#include <stdio.h>
#define MIN_YEAR 7
#define MAX_YEAR 12

int main(void) {
    printf("What year are you in? ");
    int year;
    scanf("%d", &year);
    if (MIN_YEAR <= year && year <= MAX_YEAR) {
        printf("You are in high school\n");
    } else {
        printf("You are not in high school\n");
    }
    return 0;
}
```

Code Review 2

We should declare variables right before we need them so that we don't lose track of them. **`sum`** should be declared when we sum **`num1`** and **`num2`**.

```c
#include <stdio.h>

int main(void) {
    // Naming variables
    int num1;   // num1 is the first number the user types in
    int num2;   // num2 is the second number the user types in
    int sum;    // sum is the result of adding the first and second number

    // Getting the numbers from the user
    scanf("%d", &num1);
    scanf("%d", &num2);

    // Adding the two numbers together
    sum = num1 + num2;

    // Tell the user the result
    printf("Your numbers add up to %d\n", sum);

    // Check if its divisible by 2
    if (sum %2 != 0) {
    if(num1 %2 == 0) {
    printf("The first number you've typed was even and the second number was odd\n");
    } else if (num1 %2 != 0){
    printf("The first number you've typed was odd and the second number was even\n");
    }
    } else if (sum %2 == 0) {
    if(num1 %2 == 0) {
    printf("Both the numbers you've typed were even\n");
    } else if (num1 %2 != 0){
    printf("Both the numbers you've typed were odd\n");
    }
    }

    return 0;
}
```

No header comment

Don't need to comment things that are obvious

Don't use in-line comments. Write the comments above.
Also, these comments are obvious

More comments that don't add value

whitespace!

Poor indentation

Long lines

Unnecessary checks.

Consistent commenting is good, but in general, they should **SUMMARISE** what the code will achieve and its purpose (hence why a header comment is important!), or explaining your logic, rather than transcribing the code line-by-line basis English.

Beautified 👸

```c
// A program that scans in 2 numbers from the user then
// prints their sum and their parity
#include <stdio.h>

int main(void) {
    // Scan input from user
    int num1;
    int num2;
    scanf("%d %d", &num1, &num2);

    // Tell user the sum of the numbers
    int sum = num1 + num2;
    printf("Your numbers add up to %d\n", sum);

    // Tell user the parity of the numbers.
    if (sum % 2 != 0) {
        // Their sum is odd: one number must be odd and the other is even.
        if (num1 % 2 == 0) {
            printf("The first number you've typed was even and the second "
                "number was odd\n");
        } else {
            printf("The first number you've typed was odd and the second "
                "number was even\n");
        }
    } else {
        // Their sum is even: either both numbers are odd or both are even.
        if (num1 % 2 == 0) {
            printf("Both the numbers you've typed were even\n");
        } else {
            printf("Both the numbers you've typed were odd\n");
        }
    }
    return 0;
}
```

Open question:
What other changes would you make to make the code better (aside from fixing up that grammar...)?