User-Defined Functions

COP 3223C – Introduction to Programming with C

Fall 2025

Yancy Vance Paredes, PhD

Reflection

- We have been writing **small** and **simple** programs
- We will start creating larger programs that solve **multiple tasks**
- As programs grow, some lines of code (i.e., logic) may need to be **repeated** at various places
- We need to find a way to better **organize** our codes such that we do not commit bad programming practices

Design Principles /1

- To be a good programmer, you must know **modularity**

- Consider the **Single Responsibility principle**

- This facilitates **separation of concerns** and **code reuse**

- Follow the **DRY principle** (Don’t Repeat Yourself)

Design Principles /2

- The goal is to write a program that allows for easier **debugging**

- Furthermore, we want to make **testing** easier

- Lastly, when testing programs, consider **edge cases**

Scenario

- Write a program that reads a list of `T` positive numbers
- Assume that each of these numbers represent numerical grades
- For each grade, convert it to its corresponding letter grade

Sample Run

double grade;

```
Enter T: 3
Enter #1: 90
A
Enter #2: 85
B
Enter #3: 55
F
```

Numerical	Letter
90 or above	A
80 – 90*	B
70 – 80*	C
60 – 70*	D
Below 60	F

80 - <90

70 → <80

```
if( grade >= 90 )
    printf("A");

else if ( 80 <= grade && grade < 90
    printf("B");

else
```

10 - 20 num

10 <= num && num <= 20

Program Planning

- Can you identify the **tasks** that need to be accomplished?

- What’s your strategy?

Discussion /1

- Can you imagine how complex your **`main()`** function will be?

- How?

Discussion /2

- There is an overlap among the tasks

- It’s possible that it could make debugging difficult

- Clearly, we want to improve our overall code

90 - 100

86 - 100

Discussion /3

- What if we decide to change the grade conversion table?
- How do you imagine the code will look like?

Discussion /4

- Can we *group* lines of codes that are associated with a given task?

- We give that group a **name** and **call** it to run those codes

User-Defined Functions

- A **named set of instructions** that is defined by the programmer

- If you want to use it (i.e., **invoke**) call it by its name

Types of Functions /1

- In Mathematics, there are those **that return something** (value)

- There are some that **don’t return anything** at all

Types of Functions /2

- C needs to know what **data type** the value the function will return

→ int, float, char, double

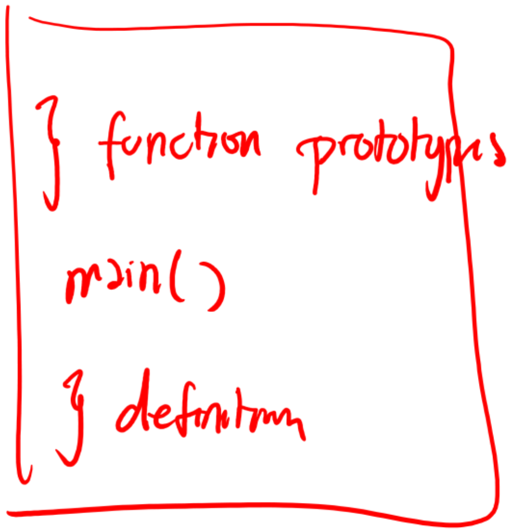
- What if the function **doesn't return** a value? What's the data type?

void

User-Defined Functions in C

- ✓ Function Prototypes
- ✓ Function Definition

Function Prototypes



- Usually placed before the **`main()`** function

- We are *declaring* a function (**Note:** There is a semicolon)

- Tells the compiler a function’s signature, such as what value it returns and what value/s it expects

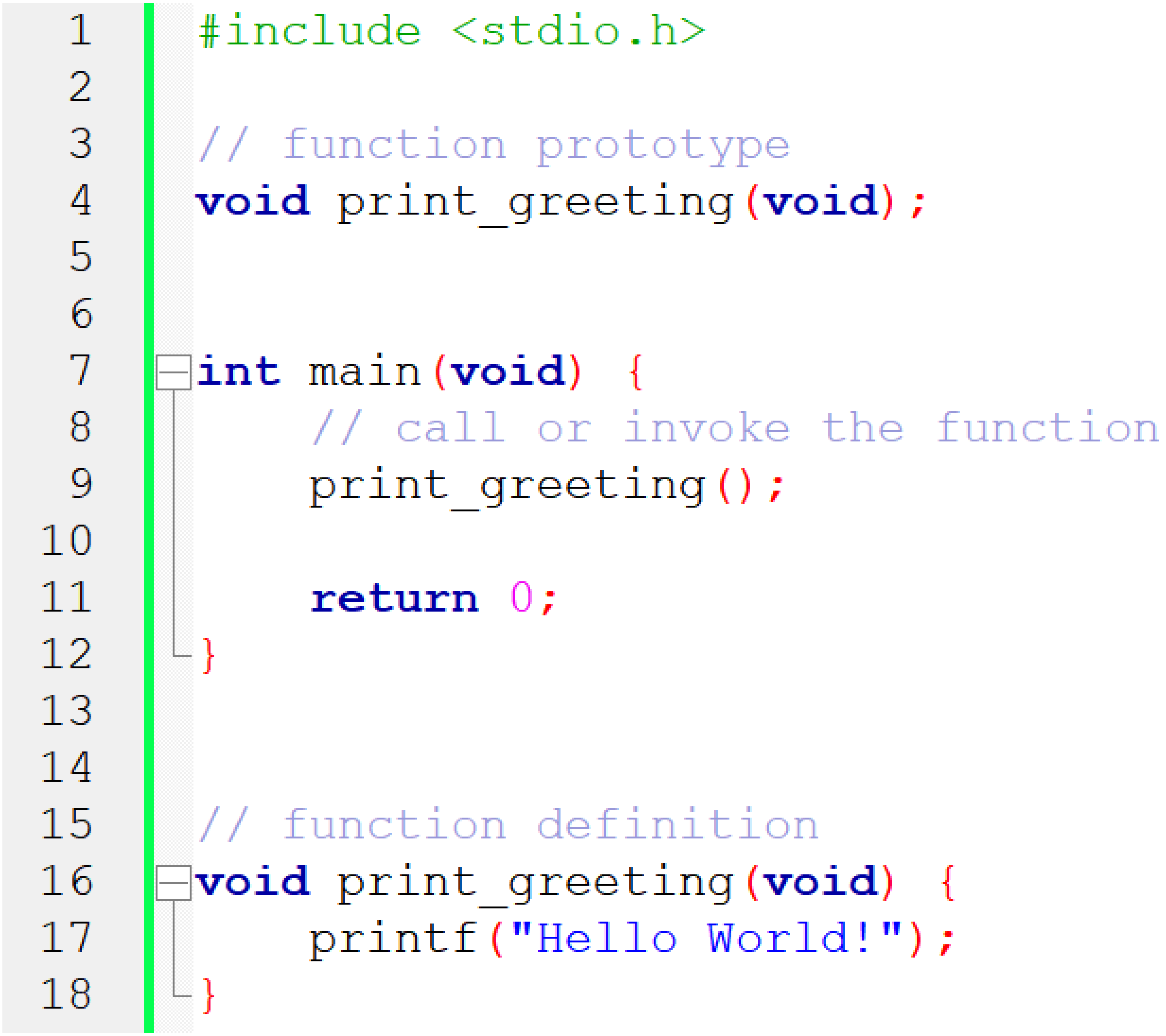
- Useful for type checking to verify the function is called correctly

Function Definition

- For a regular variable, we are creating it and *assigning* it a value
- For functions, we are specifying its behavior (i.e., what to do)
- Essentially, we are writing the *body* of the function

Practice

Write a function called **`print_greeting()`**. This function prints **`Hello World!`**

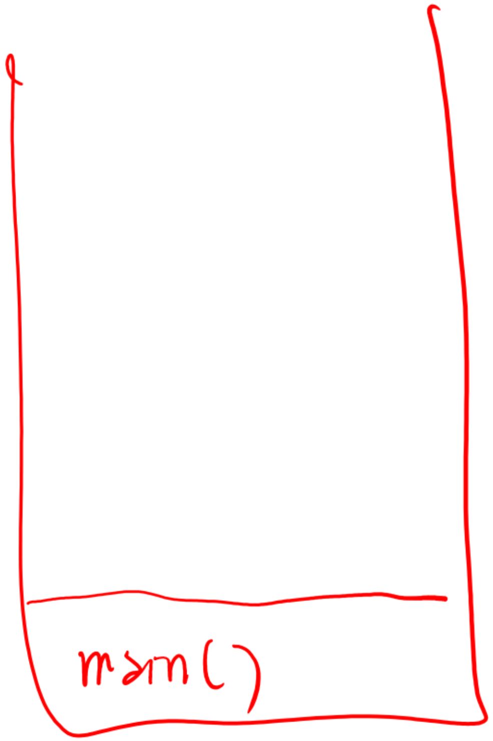


```c
#include <stdio.h>

// function prototype
void print_greeting(void);


int main(void) {
    // call or invoke the function
    print_greeting();

    return 0;
}


// function definition
void print_greeting(void) {
    printf("Hello World!");
}
```

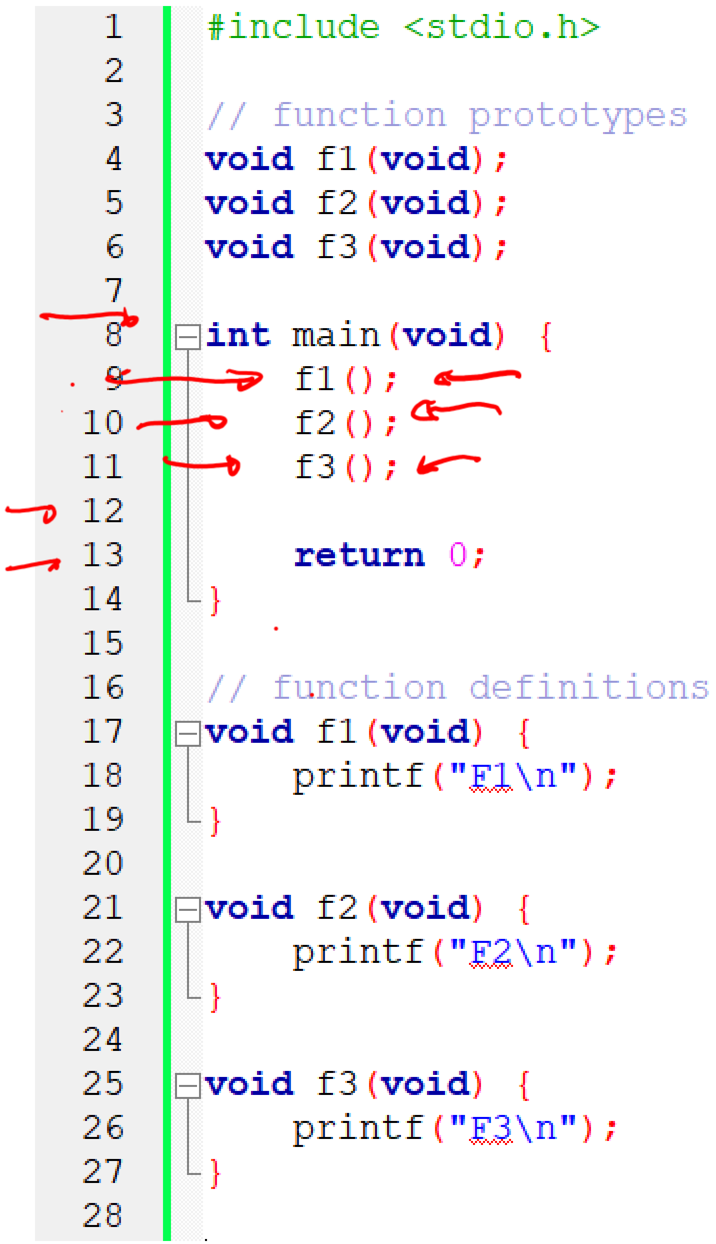
Code Tracing /1

What is the output?

F1
F2
F3



call stack



```
#include <stdio.h>

// function prototypes
void f1(void);
void f2(void);
void f3(void);

int main(void) {
    f1();
    f2();
    f3();

    return 0;
}

// function definitions
void f1(void) {
    printf("F1\n");
}

void f2(void) {
    printf("F2\n");
}

void f3(void) {
    printf("F3\n");
}
```

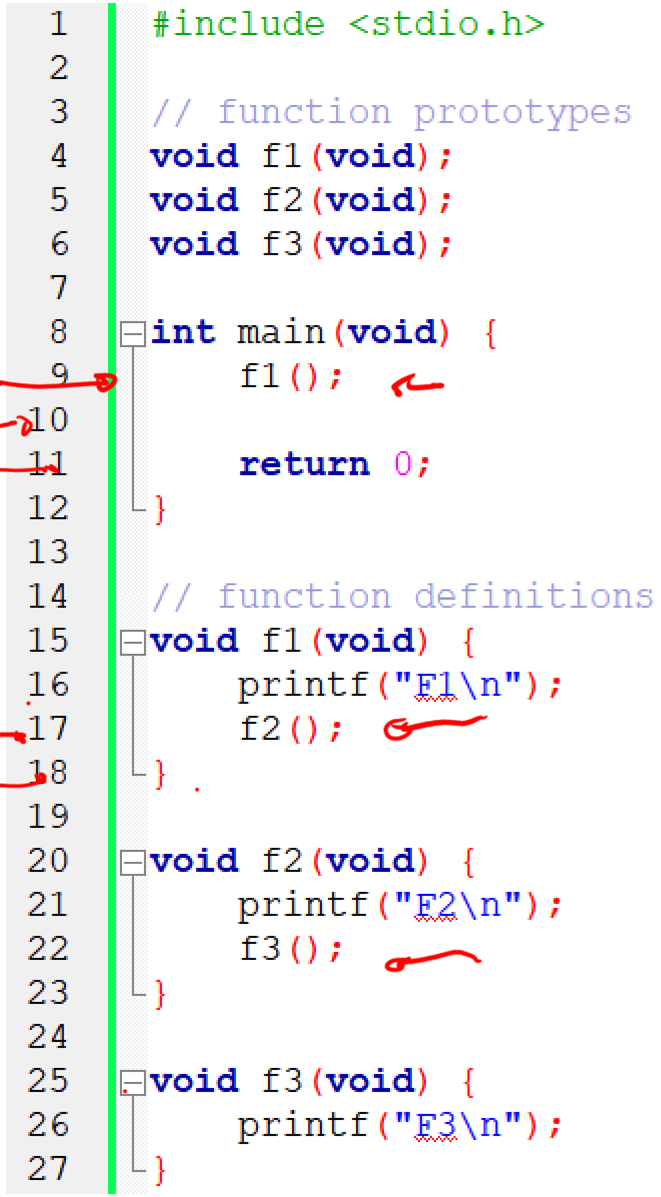
Code Tracing /2

What is the output?

F1
F2
F3

~~main()~~

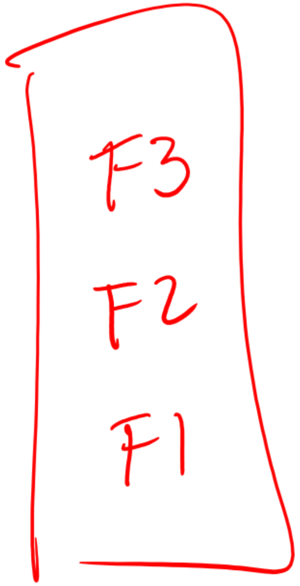
@9

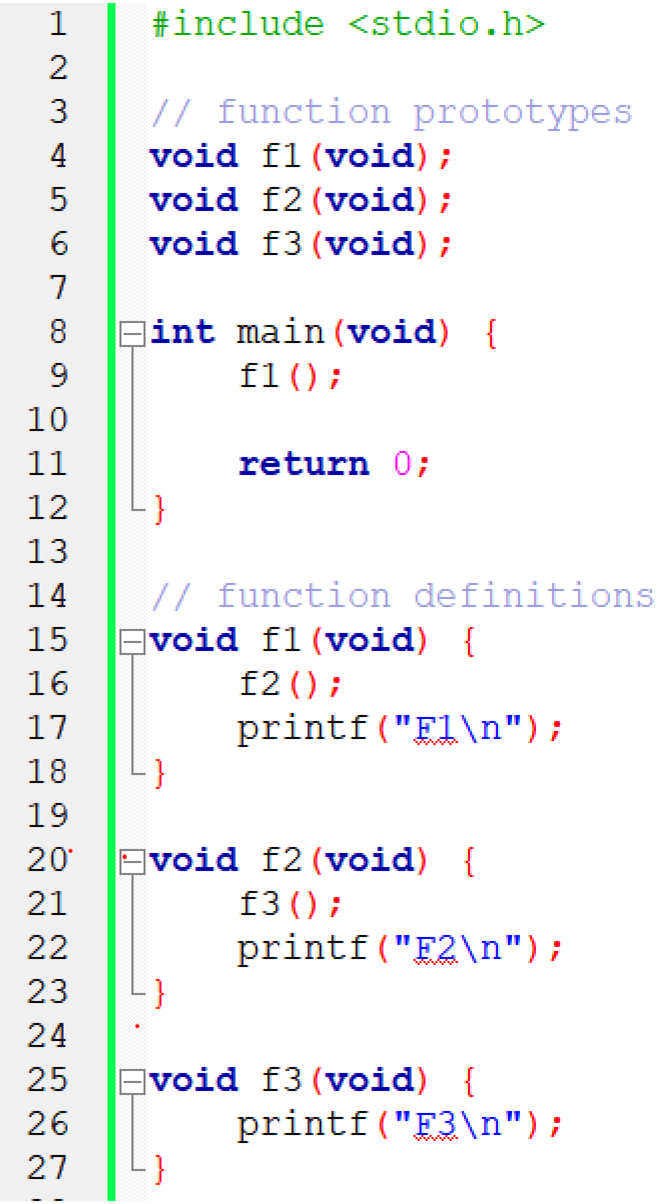


```c
#include <stdio.h>

// function prototypes
void f1(void);
void f2(void);
void f3(void);

int main(void) {
    f1();

    return 0;
}

// function definitions
void f1(void) {
    printf("F1\n");
    f2();
}

void f2(void) {
    printf("F2\n");
    f3();
}

void f3(void) {
    printf("F3\n");
}
```

Code Tracing /3

What is the output?





```c
#include <stdio.h>

// function prototypes
void f1(void);
void f2(void);
void f3(void);

int main(void)  {
    f1();

    return 0;
}

// function definitions
void f1(void)  {
    f2();
    printf("F1\n");
}

void f2(void)  {
    f3();
    printf("F2\n");
}

void f3(void)  {
    printf("F3\n");
}
```

Notes /1

- When a function calls another function, the caller **pauses**

- Control then goes to the **called function**

- Once the called function is done, control **goes back to the caller**

- It then **resumes** to where it paused

Notes /2

- Keep track of things by visualizing the contents of the **call stack**

- Especially when the called functions have their **own variables***

Discussion /1

- So far, what we've seen are functions that prints on the screen
- They were all instances of **`void`** functions
- They **did** something but **did not return** anything
- **Note:** Sometimes you may see the **`return`** statement

Discussion /2

- There are certain instances where you want to do something
- Afterward, you want to return a value to the calling function
- Think of it as *passing a message* **from callee to caller**

Functions that Return a Value

- You can only return **one value** (i.e., single)

- In your function definition, you use the **`return`** statement (`;`)

- It is followed by the value that you want to return

Discussion

What determines which data type you can return?

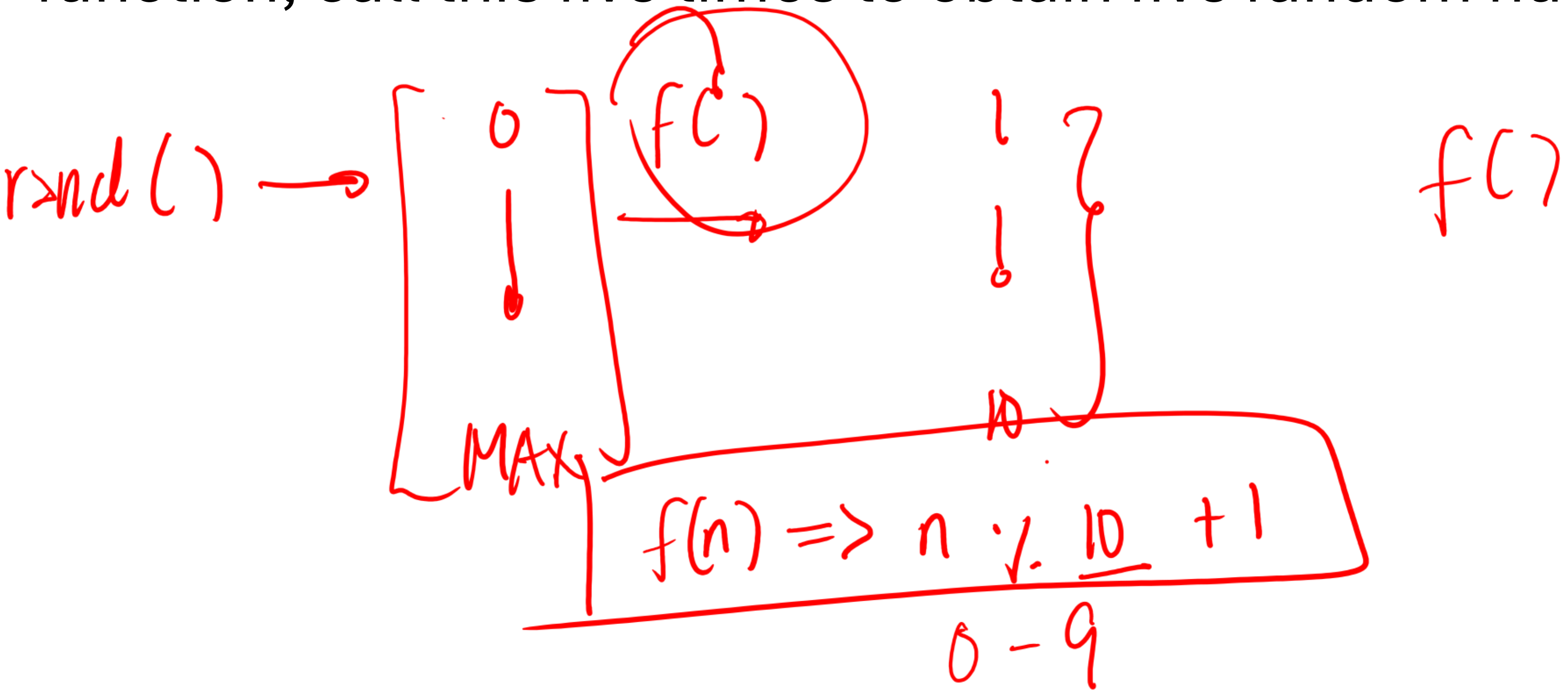
Practice

$n \% \overset{m}{20} = 0 \to 19$

$1000 \% 20$

$0 \to m-1$

Write a function called **`get_random_number()`**. It returns a random number between 1 and 10, inclusive. From the **`main()`** function, call this five times to obtain five random numbers.



Notes

- When we call the function we defined, we are expecting that it will return a value

- What is the data type of the return value?

- If you want to store the return value, you must use the **assignment operator**

Discussion

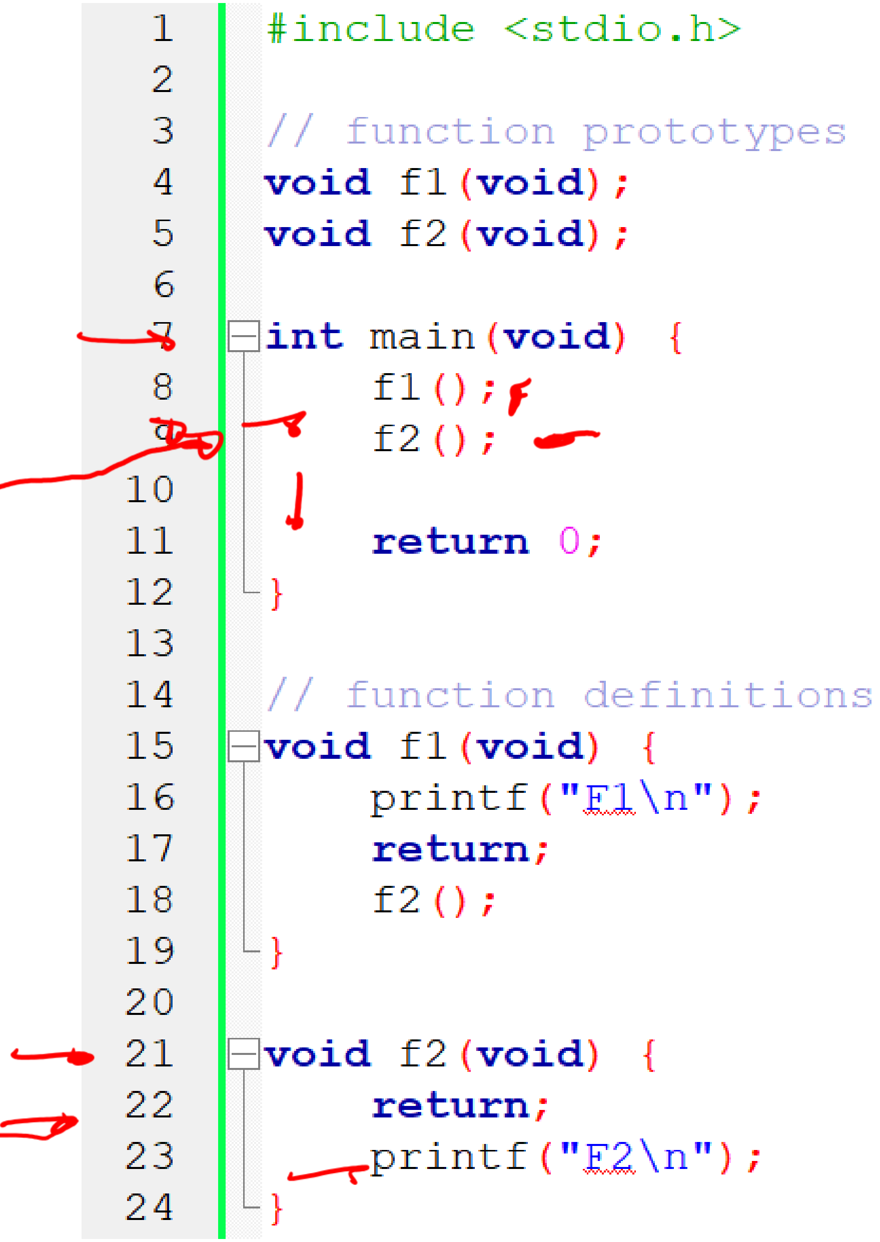
- For a non-**`void`** function, we use the **`return`** statement
- Can we also use it for **`void`** functions?

Code Tracing

F1

What is the output?

F1



```c
#include <stdio.h>

// function prototypes
void f1(void);
void f2(void);

int main(void) {
    f1();
    f2();

    return 0;
}

// function definitions
void f1(void) {
    printf("F1\n");
    return;
    f2();
}

void f2(void) {
    return;
    printf("F2\n");
}
```

Notes

- When the **`return`** statement is encountered, the remaining statements are **ignored**

- The control goes back to the calling function (i.e., caller)

Discussion

- In our previous example, we declared a variable inside the **`get_random_number()`** function

- We also declared a variable in the **`main()`** function

- Were they the same? How to confirm?

Variable Scopes /1

- Refers to a **specific region** or part of the code where a variable can be accessed and used (i.e., seen)

- Begins at the line where a variable was **declared**

Variable Scopes /2

Local Scope

Declared inside a function and visible only within the function

Global Scope (avoid)

Declared outside a function and visible to all functions

Variable Scopes /3

int i = 0

for ( ) {

}

Block Scope

Declared within a **{ }** and visible only within that block

Example: the counter variable of **`for`** loops!

i

#1

```
for (int i = 0; i < 5; i++) {
}
```

variable i only exists inside this block →

→ it is lost at this point! ←

i

#2

```
int num;
if (num > 0) {
    int tmp;
}
else {
    // tmp is not visible
    // here
}
```

variable tmp only exists inside this block →

two examples of block scopes that you may commonly encounter.

Code Tracing

10

What is the output?

F1

Enter Number: 10

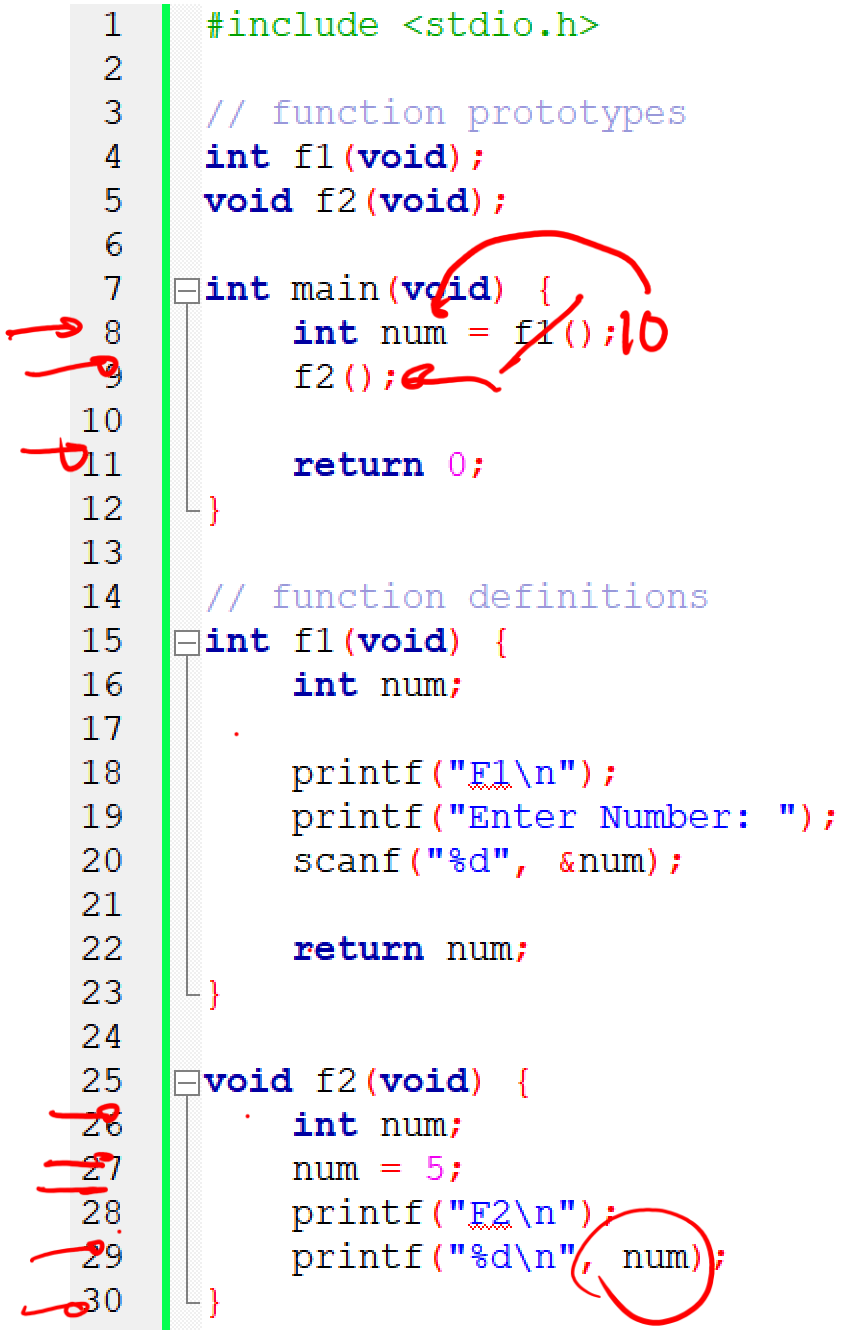
F2

5

~~f2() num=5~~

~~main() num=10~~

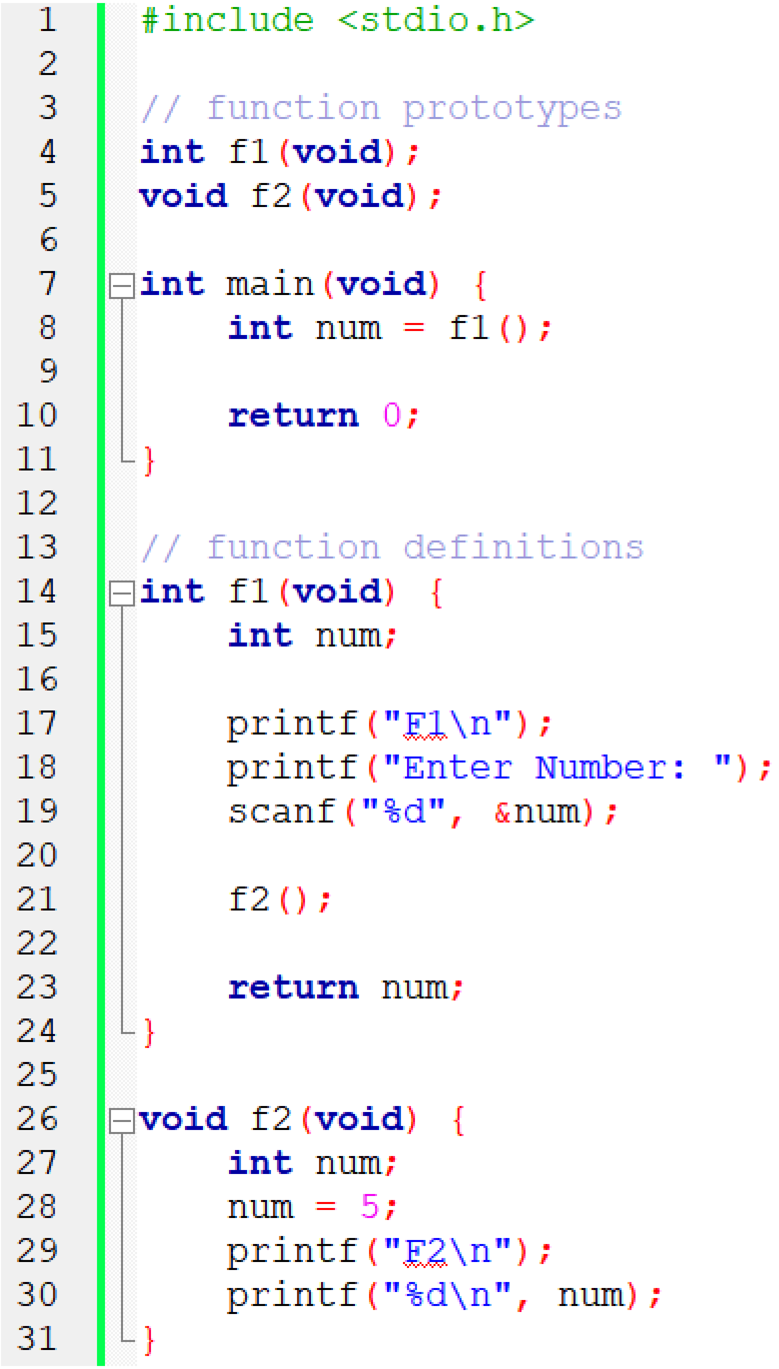
@9



```c
#include <stdio.h>

// function prototypes
int f1(void);
void f2(void);

int main(void) {
    int num = f1();
    f2();

    return 0;
}

// function definitions
int f1(void) {
    int num;

    printf("F1\n");
    printf("Enter Number: ");
    scanf("%d", &num);

    return num;
}

void f2(void) {
    int num;
    num = 5;
    printf("F2\n");
    printf("%d\n", num);
}
```

Your Turn!

What is the output?



```c
#include <stdio.h>

// function prototypes
int f1(void);
void f2(void);

int main(void) {
    int num = f1();

    return 0;
}

// function definitions
int f1(void) {
    int num;

    printf("F1\n");
    printf("Enter Number: ");
    scanf("%d", &num);

    f2();

    return num;
}

void f2(void) {
    int num;
    num = 5;
    printf("F2\n");
    printf("%d\n", num);
}
```

Recall: Built-In `math.h` Functions

The built-in functions we used needed **something** for them to work

Functions that Accept Values /1

- Formally known as **arguments** or **parameters**
- Think of them as **inputs** of the function
- Or *passing a message* **from caller to callee**

Functions that Accept Values /2

- For example, think of the **`sqrt(x)`** function in Mathematics

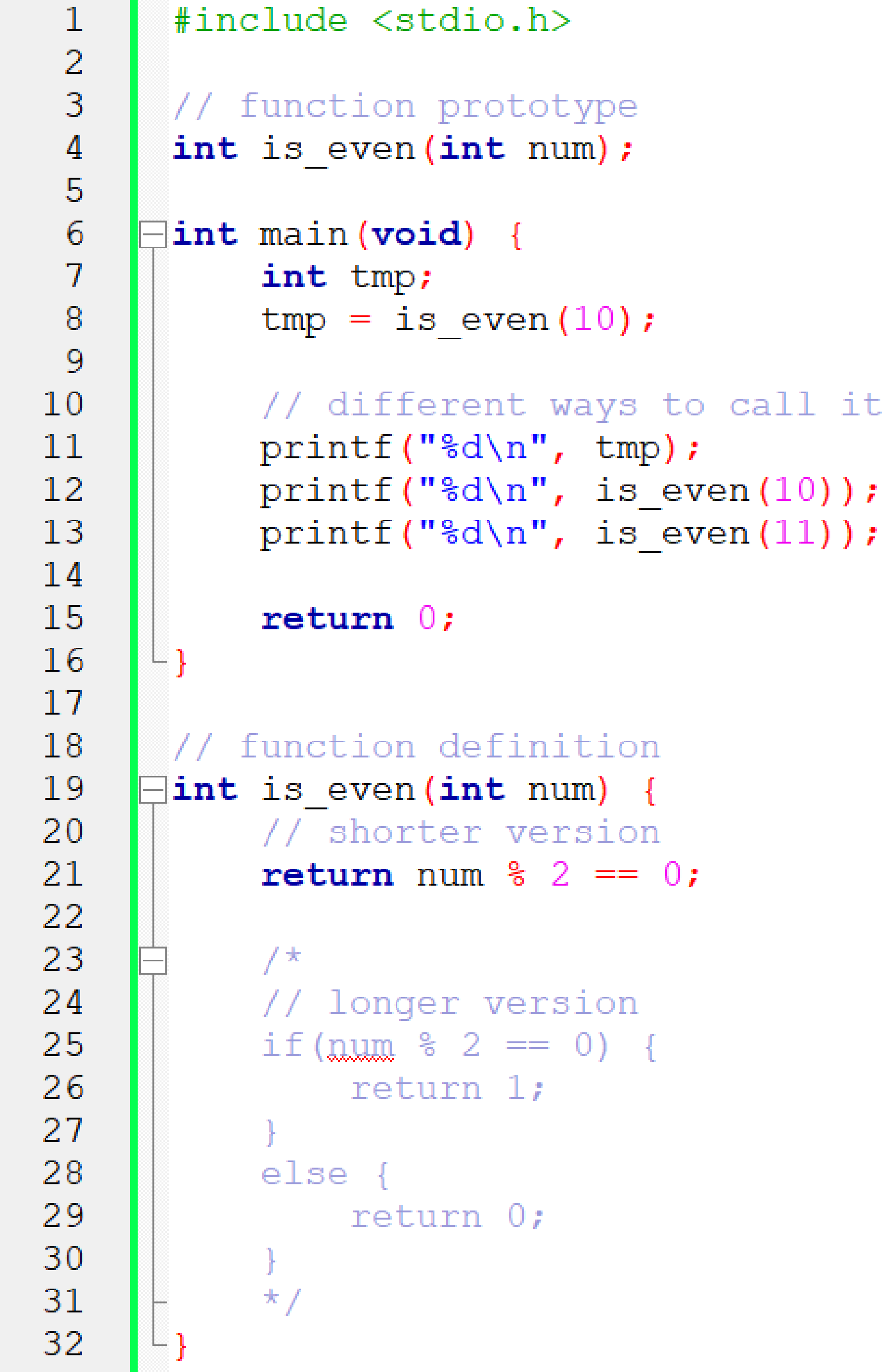
- Another example is the **`pow(a, b)`** function 2^5

Functions that Accept Values /3

- Think of the parameters as your typical local variables
- What is the scope of these variables?
- You can have multiple of these!
- **Note:** The order matters!

Practice

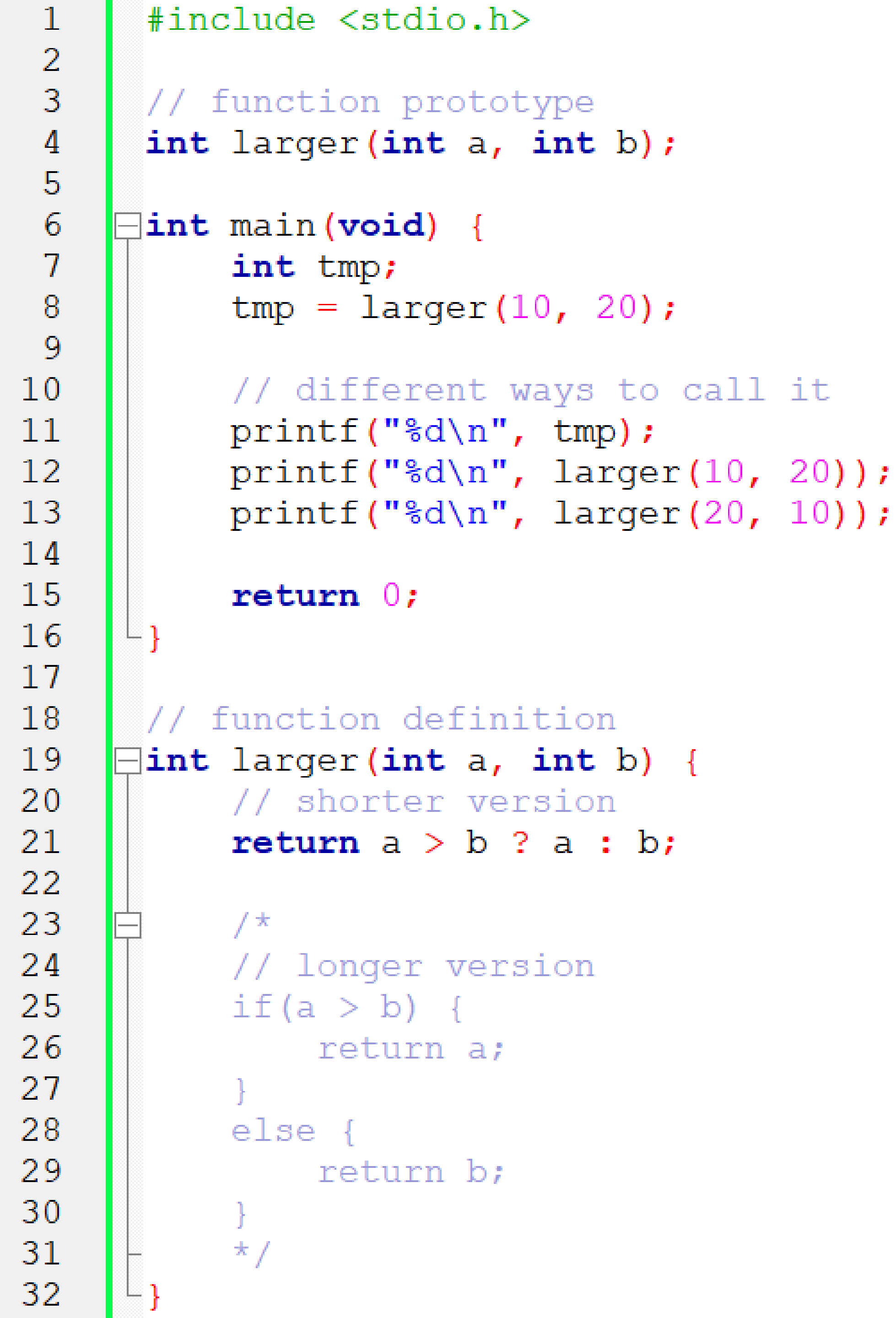
Write a function called **`is_even(num)`** which accepts a whole number **`num`**. It then returns a **`1`** if **`num`** is an **even number**. Otherwise, it returns a **`0`**.



```c
#include <stdio.h>

// function prototype
int is_even(int num);

int main(void) {
    int tmp;
    tmp = is_even(10);

    // different ways to call it
    printf("%d\n", tmp);
    printf("%d\n", is_even(10));
    printf("%d\n", is_even(11));

    return 0;
}

// function definition
int is_even(int num) {
    // shorter version
    return num % 2 == 0;

    /*
    // longer version
    if(num % 2 == 0) {
        return 1;
    }
    else {
        return 0;
    }
    */
}
```

Practice

Write a function called **`larger(a, b)`** which accepts two **distinct** whole numbers **`a`** and **`b`**. It then returns the **larger value** between the two numbers.

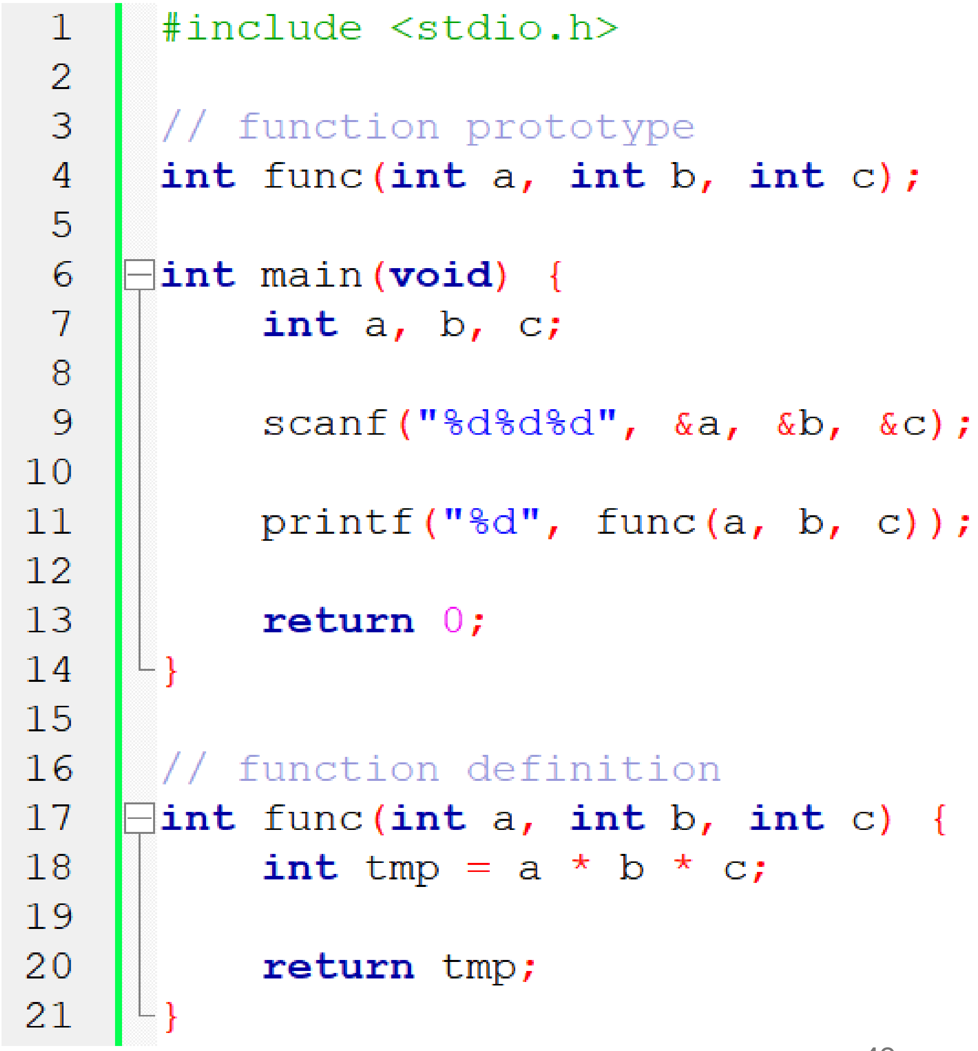


```c
#include <stdio.h>

// function prototype
int larger(int a, int b);

int main(void) {
    int tmp;
    tmp = larger(10, 20);

    // different ways to call it
    printf("%d\n", tmp);
    printf("%d\n", larger(10, 20));
    printf("%d\n", larger(20, 10));

    return 0;
}

// function definition
int larger(int a, int b) {
    // shorter version
    return a > b ? a : b;

    /*
    // longer version
    if(a > b) {
        return a;
    }
    else {
        return b;
    }
    */
}
```

Code Comprehension

What does the function **`func()`** do?

What does the entire program do?



```c
#include <stdio.h>

// function prototype
int func(int a, int b, int c);

int main(void) {
    int a, b, c;

    scanf("%d%d%d", &a, &b, &c);

    printf("%d", func(a, b, c));

    return 0;
}

// function definition
int func(int a, int b, int c) {
    int tmp = a * b * c;

    return tmp;
}
```

Terminology /1

Parameters (Formal Parameters)

The variables declared in the function definition

Think: Receiver

Arguments (Actual Parameters)

The variables or values passed to the function

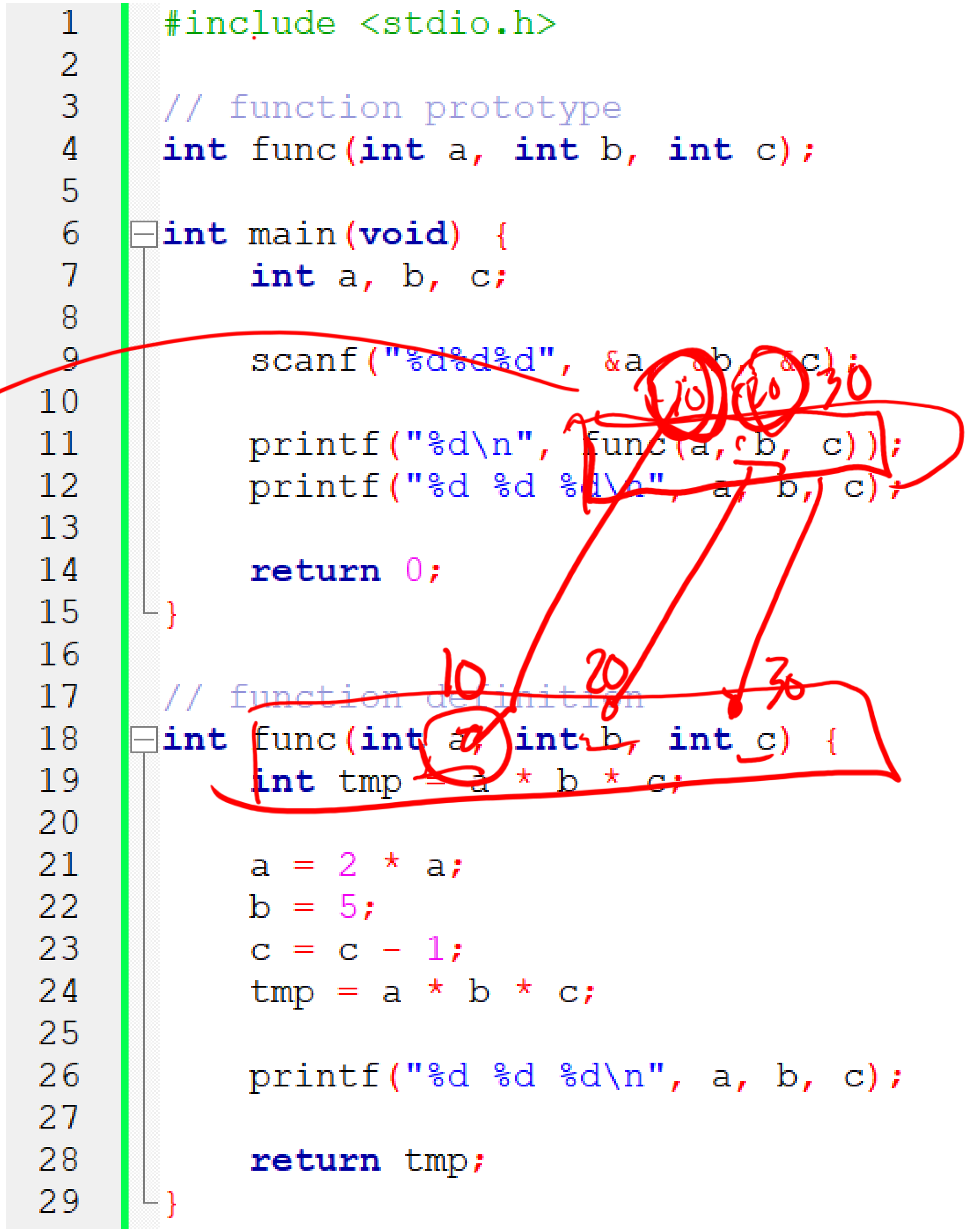
Think: Sender

Terminology /2

Which is which?

arguments /
actual parameters

parameters /
formal parameters



```
#include <stdio.h>

// function prototype
int func(int a, int b, int c);

int main(void) {
    int a, b, c;

    scanf("%d%d%d", &a, &b, &c);

    printf("%d\n", func(a, b, c));
    printf("%d %d %d\n", a, b, c);

    return 0;
}

// function definition
int func(int a, int b, int c) {
    int tmp = a * b * c;

    a = 2 * a;
    b = 5;
    c = c - 1;
    tmp = a * b * c;

    printf("%d %d %d\n", a, b, c);

    return tmp;
}
```

Code Tracing

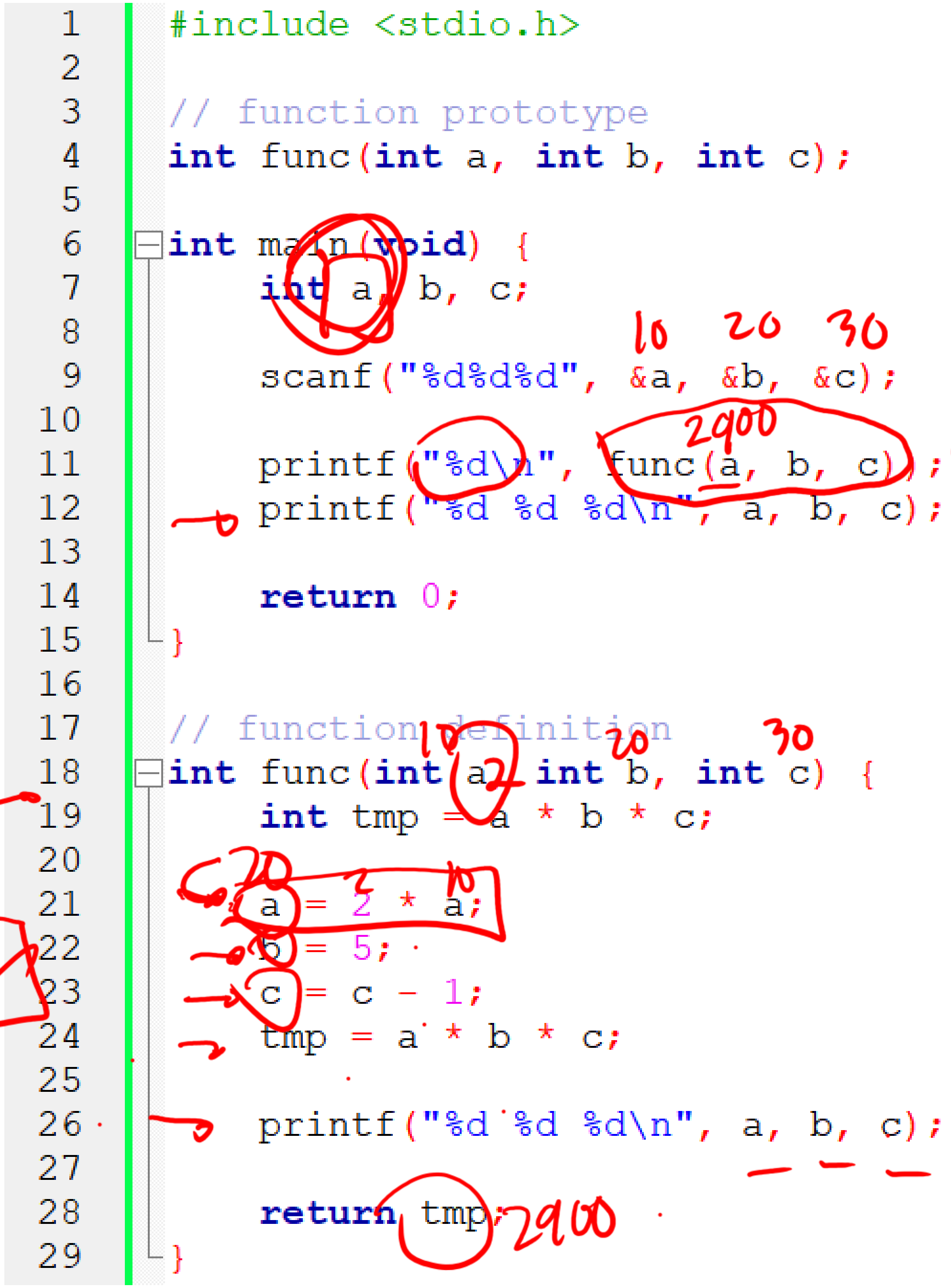
What is the output?

10 20 30

```
20 5 29
2900
10 20 30
```

func() a = 20, b = 5, c = 29, tmp = 2900

main() a = 10, b = 20, c = 30 @11



```c
#include <stdio.h>

// function prototype
int func(int a, int b, int c);

int main(void) {
    int a, b, c;

    scanf("%d%d%d", &a, &b, &c);

    printf("%d\n", func(a, b, c));
    printf("%d %d %d\n", a, b, c);

    return 0;
}

// function definition
int func(int a, int b, int c) {
    int tmp = a * b * c;

    a = 2 * a;
    b = 5;
    c = c - 1;
    tmp = a * b * c;

    printf("%d %d %d\n", a, b, c);

    return tmp;
}
```

Notes /1

- Notice what happens when functions have parameters

- What happens when the calling function modifies the variable?

Notes /2

- What happened is referred to as **pass by value**
- The called function **receives a copy** of the values passed to it
- Therefore, any “changes” done by the called function will not affect those in the calling function*

Notes /3

Once the function is done, these local variables are automatically removed from the memory

Discussion

prmt vs return

	`void` Function	Non-**`void`** Function
No Arguments	Print something `void print_greeting(void)`	Do something and return something `int get_random_number(void)`
With Arguments	Print something using the passed value `void print_message(int count)`	Mostly computation and returns the computed value `int pow(a, b)`

'a' 'f'

Practice – Function Completion

Complete the definition of function **`count_vowels()`**. This function accepts two letters from the English alphabet. Afterward, it returns the **number of vowels** in between them (inclusive). You may only use built-in functions. You cannot define any other functions.

count_vowels('a', 'e') → 2

count_vowels('A', 'f') → 2

```
int count_vowels(char start, char end) {

}
```

count_vowels('z', 'A') → 5.

char alpha

alpha == 'a' || alpha == 'e'.

('A', 'a') ⇒ 1

Discussion

```
if( start > end)
```

- Notice the code that **swaps** the values of two variables?
- It is a common logic to use a third variable (i.e., **temporary**)
- You’ll see more when dealing with sorting algorithms*

```
tmp = start
start = end
end = tmp
```

```
start = 'z'        start = 'a'
end = 'a'    ⇒     end = 'z'
```

Challenge

Complete the definition of function **`get_reverse()`**. This function accepts a nonnegative whole number **`num`**. Afterward, it returns the **reverse** of **`num`**. For example, **`get_reverse(123)`** will return **`321`**. You cannot define any other functions.

```
int get_reverse(int num) {



}
```

Questions?