CS 137

Functions, Modules, and Compiling

Fall 2025

Victoria Sakhnini

Table of Contents

Working with Functions

We've already seen functions; `int main(void)` is a function!

Syntax:

```
return-type fun_name(parameter(s)) {function body / statements}
```

A function is a group of statements that together perform a task. Every C program has at least one function, which is `main()`.

The `return-type` must be specified in C99, and later. It is `void` if the function does not return anything.

`fun_name` follows the same rules as variable names.

`parameter(s)` (if exist) must have their type declared per variable/parameter, e.g. `int fun(int a, int b)` (`int fun(int a, b)` is incorrect)

The parameters are local variables used only inside the function body. We will learn more about the scope of a variable in a bit.

`return` (if it exists) ends the function and returns the value after it. Otherwise, when we reach the end of the function (`}`), it returns to the caller.

For calling a function with no parameters, you use empty brackets: `fun_name();`

Basic example:

```
1. # include <stdio.h>
2. int max (int a, int b)
3. {
4.        return a > b ? a : b;
5. }
6.
7. int main (void)
8. {
9.     printf("%d", max (5 ,10));
10.    return 0;
11.  }
```

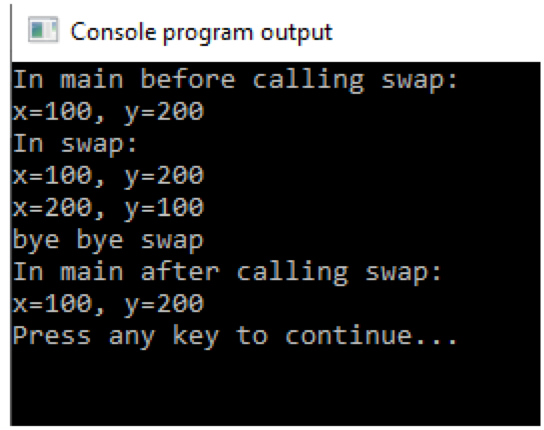
Steps:

- The execution always starts with the `main` function.
- To execute `printf` in line 9, `max(5, 10)` is called (5 and 10 are called the arguments), we have two arguments because the function `max` was defined with two parameters `a` and `b` of type `int`.
- When the function `max` is called, the value 5 is assigned to `a`, and the value 10 is assigned to `b`.
- Then, the function body is executed. since `a > b` is false (5>10 is false) then the returned value is 10 (the value of `b`).
- This value (10) is returned to the same place where the function was called; thus, `printf` in line 9 will output 10,
- then the `main` function returns 0, which ends the program.

This function call is by value, which means a copy of the arguments' values is assigned to the parameters. (We will understand what this means and what other options are available in the future.)

For another example, let us consider the following program and the output:

```
#include <stdio.h>

void swap(int x, int y)
{
      printf("In swap:\n");
      printf("x=%d, y=%d\n", x, y);
      int temp;
      temp = x;
      x = y;
      y = temp;
      printf("x=%d, y=%d\n", x, y);
      printf("bye bye swap\n");
  }
  int main(void)
  {
       int x = 100;
       int y = 200;
       printf("In main before calling swap:\n");
       printf("x=%d, y=%d\n", x, y);
  // calling swap
       swap(x, y);
  // returning from swap
       printf("In main after calling swap:\n");
       printf("x=%d, y=%d\n", x, y);

       return (0);
  }
```



Note that the variables x and y in the main are not the same as the variables x and y in the swap; they have the same names, but each has its own space in memory. The x and y defined in the main can be accessed only in the main body. The x and y defined in swap can be accessed only in swap body

Trace table:

x (in main)	y (in main)	x (in swap)	y (in swap)	temp (in swap)	output
100					
	200				
					In main before calling swap:
					x=100, y=200
		100			
			200		
					In swap:
					x=100, y=200
				100	
		200			
			100		
					x=200, y=100
					bye bye swap[1]
					In main after calling swap:
					x=100, y=200

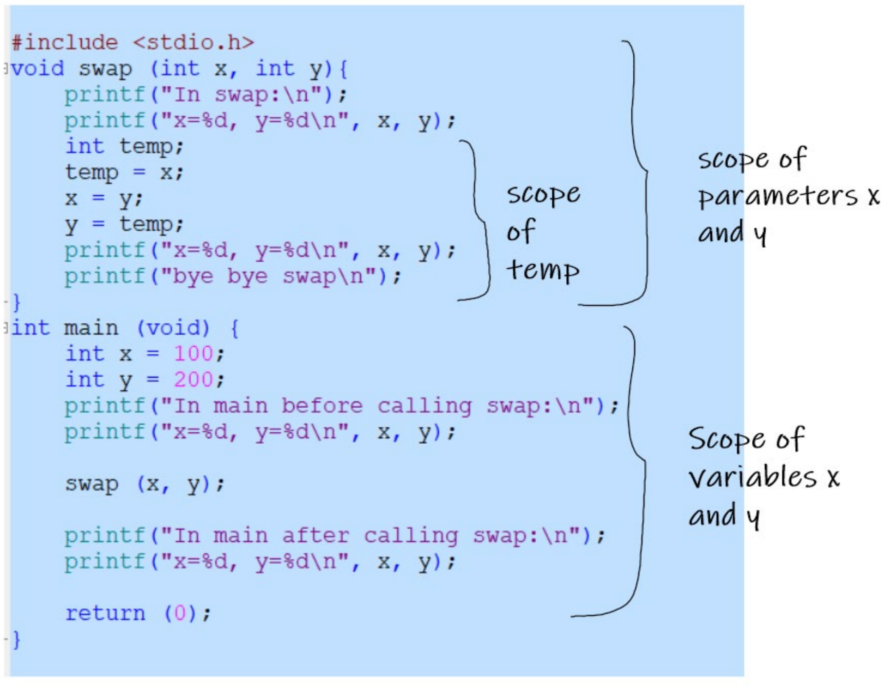
[1] This is the last statement in swap, after we return to main we can not access x and y in swap anymore.

Scope of variables

The variables inside the function `swap` are what we call local variables. The values from `main` are copied into the local variables in `swap` function. Changing the local variables inside `swap` does not affect the variables in `main` (Also, returning a local variable returns a copy of the value, which we will see examples of later).

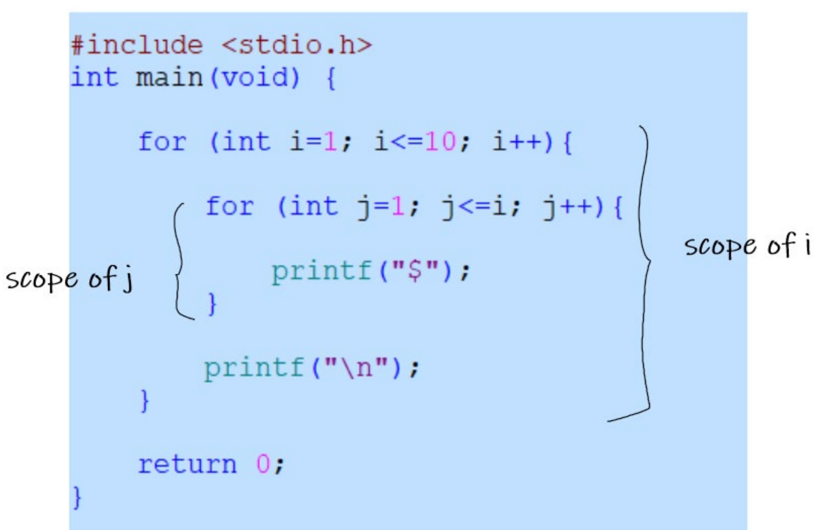
A scope in programming is the section of the program where a defined variable exists; beyond that section, it can not be accessed.

Example1:




```c
#include <stdio.h>
void swap (int x, int y){
    printf("In swap:\n");
    printf("x=%d, y=%d\n", x, y);
    int temp;
    temp = x;
    x = y;
    y = temp;
    printf("x=%d, y=%d\n", x, y);
    printf("bye bye swap\n");
}
int main (void) {
    int x = 100;
    int y = 200;
    printf("In main before calling swap:\n");
    printf("x=%d, y=%d\n", x, y);

    swap (x, y);

    printf("In main after calling swap:\n");
    printf("x=%d, y=%d\n", x, y);

    return (0);
}
```

Example2:



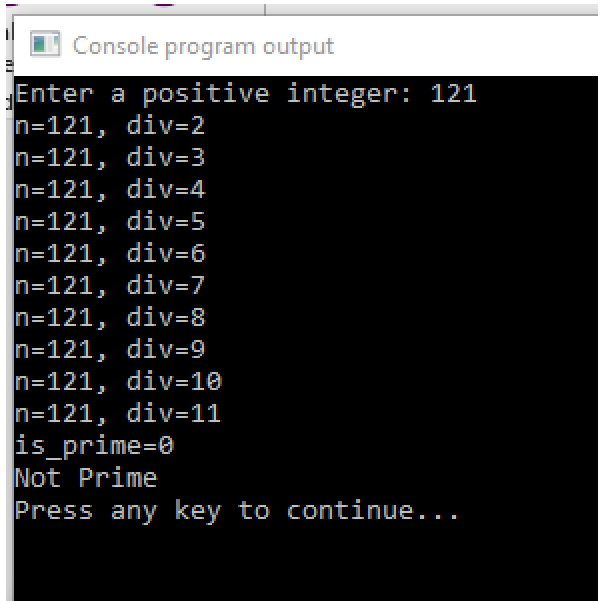

```c
#include <stdio.h>
int main(void) {

    for (int i=1; i<=10; i++){

        for (int j=1; j<=i; j++){

            printf("$");
        }

        printf("\n");
    }

    return 0;
}
```

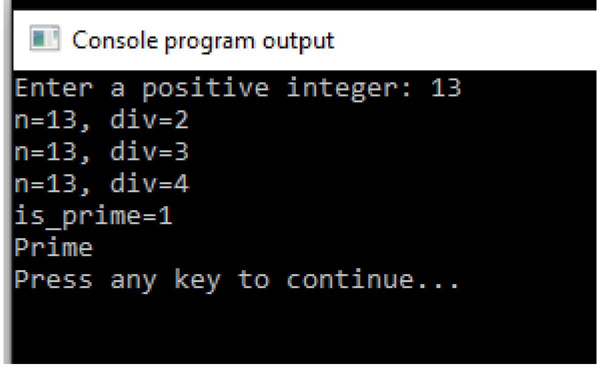
Boolean variables

One more quirk about C is that there are no boolean variables. In `C99` and later, a library `<stdbool.h>` gives you boolean variables. It turns out that these `bool` variables are secretly unsigned integers in disguise (so even with this, they aren't Boolean variables!), and these take up a full byte like a `char`. These types can only take `0` and `1` as values (all non-zero values are just `1`). You can also use the words `true` and `false` with this library.

Example:

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

bool isPrime(int n)
{
    int div = 2;
    if (n <= 1)
      return false;
  // The following print is for tracing
  // variables to understand the process
    printf("n=%d, div=%d\n", n, div);
    while (div * div <= n)
    {
        if (n % div == 0)
            return false;
        div++;
  // The following print is for tracing
  // variables to understand the process
        printf("n=%d, div=%d\n", n, div);
     }
     return true;
 }

 int main(void)
 {
     int n;
     printf("Enter a positive integer: ");
     scanf("%d", &n);
     bool is_prime = isPrime(n);
     printf("is_prime=%d\n", is_prime);
     if (is_prime)
         printf("Prime\n");
     else
         printf("Not Prime\n");
     return 0;
 }
```





`if (is_prime == true)` is equivalent to `if (is_prime)`

I've added `printf` statements to track variables to help you understand the process and the changes in values for each variable. You can find a complete trace for 121 and 13 here [i]

Using this approach helps understand code and debugging.

Function declarations

In our example, we defined the function before using it. Strictly speaking, C doesn't force us to do this. We can include a function declaration and a promise to C that we'll eventually define this function with this given return type. The declaration is the first line of the function but ends with a semicolon. The declaration may not include the parameters, though it is advised to do so.

The previous example can also be written in the following way:

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

bool isPrime(int n);  // Function Declaration

int main(void)
{
    int n;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    bool is_prime = isPrime(n);
    printf("is_prime=%d\n", is_prime);
    if (is_prime)
          printf("Prime\n");
    else
          printf("Not Prime\n");
    return 0;
}

bool isPrime(int n)
{
    int div = 2;
    if (n <= 1)
      return false;
        // The following print is for tracing
// variables to understand the process
    printf("n=%d, div=%d\n", n, div);
    while (div * div <= n)
    {
        if (n % div == 0)
            return false;
        div++;
// The following print is for tracing
// variables to understand the process
          printf("n=%d, div=%d\n", n, div);
    }
    return true;
}
```

Some programmers prefer to declare the functions before `main` and write the implementation after `main`.

Either way, **the function must be declared or defined before another function calls it**.

assert

Let us look at the following problem.

The Gregorian calendar replaced the Julian calendar in most of Europe in 1582. North America (i.e., England and its colonies) adopted this calendar in September 1752. A leap year contains an extra day that occurs every four years, not multiples of 100 unless they are also multiples of 400.
Examples: 2016, 2000, 1804 were all leap years.
Non-Examples: 2017, 1900, 1950 were not leap years.
Write a function is_leap_year that returns true if a given year was a leap year and false otherwise.

Solution:

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

bool is_leap_year(int year)
{
  if ((((year % 4) == 0) && ((year % 100) != 0)) || (year % 400) == 0)
          return true;
  else
          return false;
 }

```

This works well but gives us some problems if the year were to be negative. Since we are in North America, we want the year to be at least 1752. We can accomplish this by using assert statements. First, add #include <assert.h> to the beginning then include assert(year > 1752);

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

bool is_leap_year(int year)
{
    assert(year > 1752);
    if ((((year % 4) == 0) && ((year % 100) != 0)) || (year % 400) == 0)
          return true;
    else
          return false;
 }

 int main(void)
 {
    int year;
    printf("Enter a year: ");
    scanf("%d", &year);
    if (is_leap_year(year))
         printf("It is a leap year\n");
    else
         printf("It is not a leap year\n");
 }

```

```
@ubuntu1804-008% gcc leap.c
@ubuntu1804-008% ls
a.out  leap.c
@ubuntu1804-008% ./a.out
Enter a year: 234
a.out: leap.c:6: is_leap_year: Assertion `year > 1752' failed.
Aborted
@ubuntu1804-008% ./a.out
Enter a year: 1800
It is not a leap year
```

In general, `assert(expr);` If `expr` is `true`, this line does **nothing.** Otherwise, it **terminates the program** with a filename, line number, function, and expression message. This is great for debugging. It's also great to leave it in (so long as `expr` is not computationally expensive). It helps to remember assumptions, causes the program to fail "*loudly*" vs "*quietly*", and advises other programmers if the code undergoes modifications. Finally, it is suitable for regression testing, checking that changes haven't broken anything in another part of the code.

Working with Modules and Compiling

In the "real world", programs are coded by many programmers. It is often inefficient for them to all work on the same file, and it can get very confusing when you have millions of lines of code. Therefore, we want to modularize the design and reduce compile time.

Modular programming divides the program into sub-programs, each serving a specific goal. Breaking the large program into small problems increases the program's readability and maintainability and the reusability of the small sub-programs.

Each module has a well-defined interface that specifies what services it provides, as well as an implementation part that hides the code and other details from the user (by providing an executable file to the user so they can't read the actual implementation but can use the provided functions listed in the interface with documentation).

An additional advantage is that changing the implementation without changing the interface does not require the user to change the main program that uses/includes those modules. Also, it is much easier to debug a program this way.

```
@ubuntu1804-008% cat powers.h
#ifndef POWERS_H // Prevents multiple inclusion
#define POWERS_H

/*      Pre : num is a valid integer
        Post : returns the square of num .
*/

int square (int num );
int cube (int num );
int quartic (int num );
int quintic (int num );

#endif
```

This is the interface file (.h) that includes definitions of new data types (*will see examples later*) and function declarations.

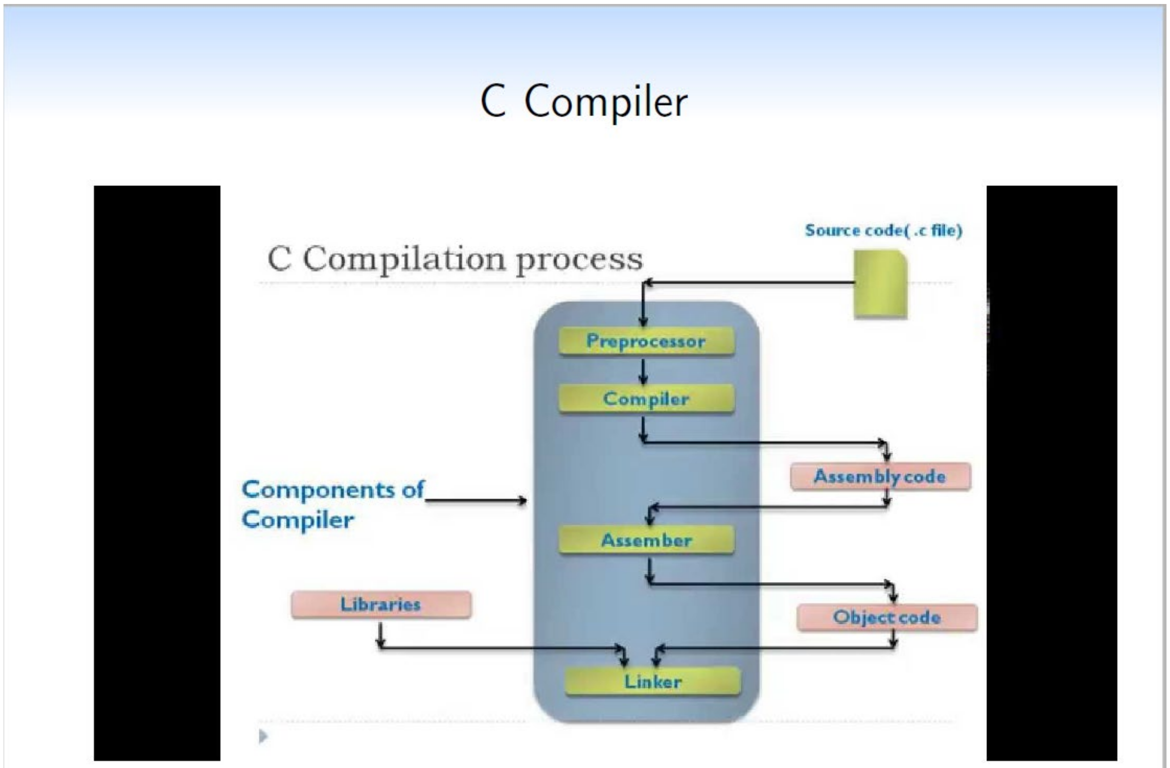
```
@ubuntu1804-008% cat powers.c
#include "powers.h" // notice the quotes !
int square (int num ) {
        return num * num ;
}

int cube (int num ) {
        return num * square (num );
}

int quartic (int num ) {
        return square (num )* square (num );
}

int quintic (int num ) {
        return square (num )* cube (num );
}
```

This is the implementation file (.c with the same name as the interface file) which includes the function implementations which were declared in the interface file. It might include helper functions needed for implementation as well (those are not declared in the interface file as users are not supposed to call them directly).

```
@ubuntu1804-008% cat prog.c
#include <stdio.h>
#include "powers.h"
int main ( void ){
        int num =3;
        printf ("%d^4 = %d\n", num , quartic (num ));
        num = 2;
        printf ("%d^5 = %d\n", num , quintic (num ));
        return 0;
}

@ubuntu1804-008% gcc powers.c prog.c
@ubuntu1804-008% ./a.out
3^4 = 81
2^5 = 32
```

Note how we compiled both files; the implementation file and the program file to link them together.

More about the C compilation process



Source:
`https://i.ytimg.com/vi/VDslRumKvRA/maxresdefault.jpg`

To compile a `C` program, we use the `gcc` compiler in Linux, which stands for **G**nu **C**ompiler **C**ollection.

It creates an executable called `a.out` (unless you request a different name). Now, I will review the details of what happens when you invoke the `C` compiler `gcc`.

When you invoke `gcc`, a series of steps are performed as indicated in the chart above!

The processor

It removes comments from the source code and interprets preprocessor directives (given by statements that begin with #, such as `#include`).

Compiling and assembling

The compiler translates the `C` code into assembly language (a machine-level code containing instructions that manipulate the memory and processor directly in a layer beneath the operating system). You do not usually see this level of compilation. Instead, you see what is known as the object code. The compiler creates the assembly code and converts the machine-level instructions into binary code. You can create object code from a `C` source with

```
$ gcc -c prog.c
```

This creates a binary file called `prog.o` that cannot be viewed with a text viewer.

Linking

The linker processes the `main` function and any possible input arguments you might use, links your program with other programs that contain functions that your program uses (libraries), and links other pre-compiled object files together to create an executable file.

More on Macros:

We've already seen three macros, namely `#include`, `#ifndef` and `#define`. In fact, we can use the `#define` to define constants in our code.

Syntax: `#define identifier replacement`

Notice that you don't need an equal sign or a semicolon; this is a straight replacement. This can be useful for constants in your code.

Example:

```c
#include <stdio.h>
#define PI 3.1415
int main(void) {
  int r = 3;
  printf("Area:
    %f", PI*r*r);
  return 0;
}
```

```c
#include <stdio.h>
int main(void) {
  int r = 3;
  printf("Area:
    %f", 3.1415*r*r);
  return 0;
}
```

Preprocessing turns the left into the right with regard to the constant. (same should happen to `#include`, but the result looks messy, so we will not show it here). BTW, you can run `gcc` on the program file with the `-E` flag to see the preprocessor output.

`3.1415` is a float value (non-integer); we will learn about this type later.

Additional Examples

```c
/*
 * Computes the number of combinations of n items taken r at a time
 */

#include <stdio.h>
int factorial(int n);

/*
 * Demonstrates multiple calls from the main function passing different
 * actual arguments to a user-defined function.
 */
int main(void)
{
        int n, r, c;

        printf("Enter total number of components> ");
        scanf("%d", &n);
        printf("Enter number of components selected> ");
        scanf("%d", &r);
        if (r <= n)
        {
                c = factorial(n) / (factorial(r) * factorial(n - r));
                printf("The number of combinations is %d\n", c);
        }
        else
        {
                printf("Components selected cannot exceed total number\n");
        }
        return (0);
}

/*
 * Computes n! for n greater than or equal to zero
 */
int factorial(int n)
{
        int i,   /* local variables */
        product = 1;

        /* Computes the product n x (n-1) x (n-2) x ... x 2 x 1 */
        for (i = n; i > 1; --i)
        {
                product *= i;
        }

        /* Returns function result */
        return (product);
}
```

Console program output

```
Enter total number of components> 5
Enter number of components selected> 2
The number of combinations is 10
Press any key to continue...
```

```
/* Finds and displays the smallest divisor (other than 1) of the integer n.
 * Displays that n is a prime number if no divisor smaller than n is found. */


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

#define NMAX 1000


int even(int num)
{
        int ans;
        ans = ((num % 2) == 0);
        return (ans);
}


int find_div(int n)
{
        int trial,       /* current candidate for smallest divisor of n       */
            divisor;     /* smallest divisor of n; zero means divisor not yet found */

        /* Chooses initialization of divisor and trial depending on whether n is even or odd.  */
        if (even(n))
                divisor = 2;
        else
        {
                divisor = 0;
                trial = 3;
        }

        /* Tests each odd integer as a divisor of n until a divisor is found this way or until
trial is so large that it is clear that n is the smallest divisor other than 1. */
        while (divisor == 0)
        {
                if (trial > sqrt(n))
                        divisor = n;
                else if ((n % trial) == 0)
                        divisor = trial;
                else
                        trial += 2;
        }

        /* Returns problem output to calling module. */
        return (divisor);
}

int main(void)
{
        int n,   /* number to check to see if it is prime         */
            min_div;     /* minimum divisor (greater than 1) of n         */

        /* Gets a number to test.         */
        printf("Enter a number between 2 and 1000 that you think is a prime number> ");
        scanf("%d", &n);
```

```c
        /* Checks that the number is in the range 2...NMAX        */
        if (n < 2)
                printf("Error: number too small. The smallest prime is 2.\n");
        else if (n <= NMAX)
        {
                /* Finds the smallest divisor (> 1) of n */
                min_div = find_div(n);
                /* Displays the smallest divisor or a message that n is prime. */
                if (min_div == n)
                        printf("%d is a prime number.\n", n);
                else
                        printf("%d is the smallest divisor of %d.\n", min_div, n);
        }
        else
                printf("Error: largest number accepted is %d.\n", NMAX);
        return (0);
}
```

Extra practice problems

[Some solutions can be found at the end of the file; however, try to solve it before reviewing my suggested solution. This is also true for all future chapters]

1) Complete the following program[ii] to be able to run it successfully. You may not include any additional interfaces.

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

int max3(int a, int b, int c);

int min3(int a, int b, int c);

// middle assumes that the three numbers are different
int middle(int a, int b, int c);

int main(void)
{

        assert(max3(9, 8, 17) == min3(234, 17, 89));
        assert(max3(9, 9, 9) == min3(9, 9, 9));
        assert(max3(19, 9, 19) == min3(99, 19, 19));

// middle assumes that the three numbers are different
        assert(middle(14, 33, 10 ) == 14);
        assert(middle(114, 33, 10) == 33);

        printf("Good job\n");

        return 0;
}
```

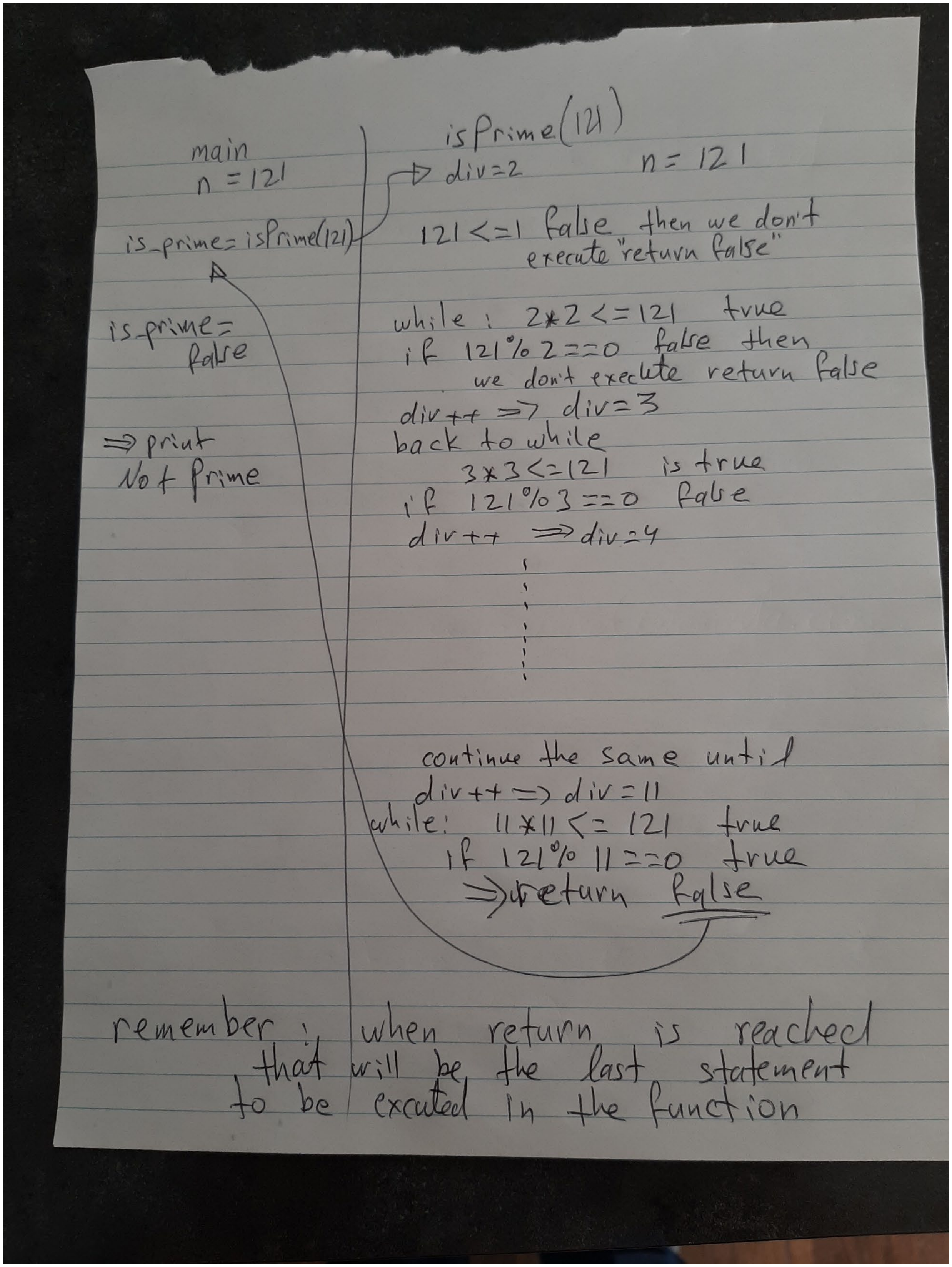
2) Consider the following interface file `funumbers.h` (You may not change this file at all)

```c
#ifndef FUNUMBERS_H // Prevents multiple inclusion
#define FUNUMBERS_H
#include <stdbool.h>

// Pre: num is a valid positive integer
// the function returns true if num and its revered digits are equivalent
// examples: 12321     555    7
// non-examples: 345    15
bool is_palindrome(int num);

 // the function returns the biggest prime number that is smaller than num
 // num > 2
 int big_prime(int num);

 #endif
```

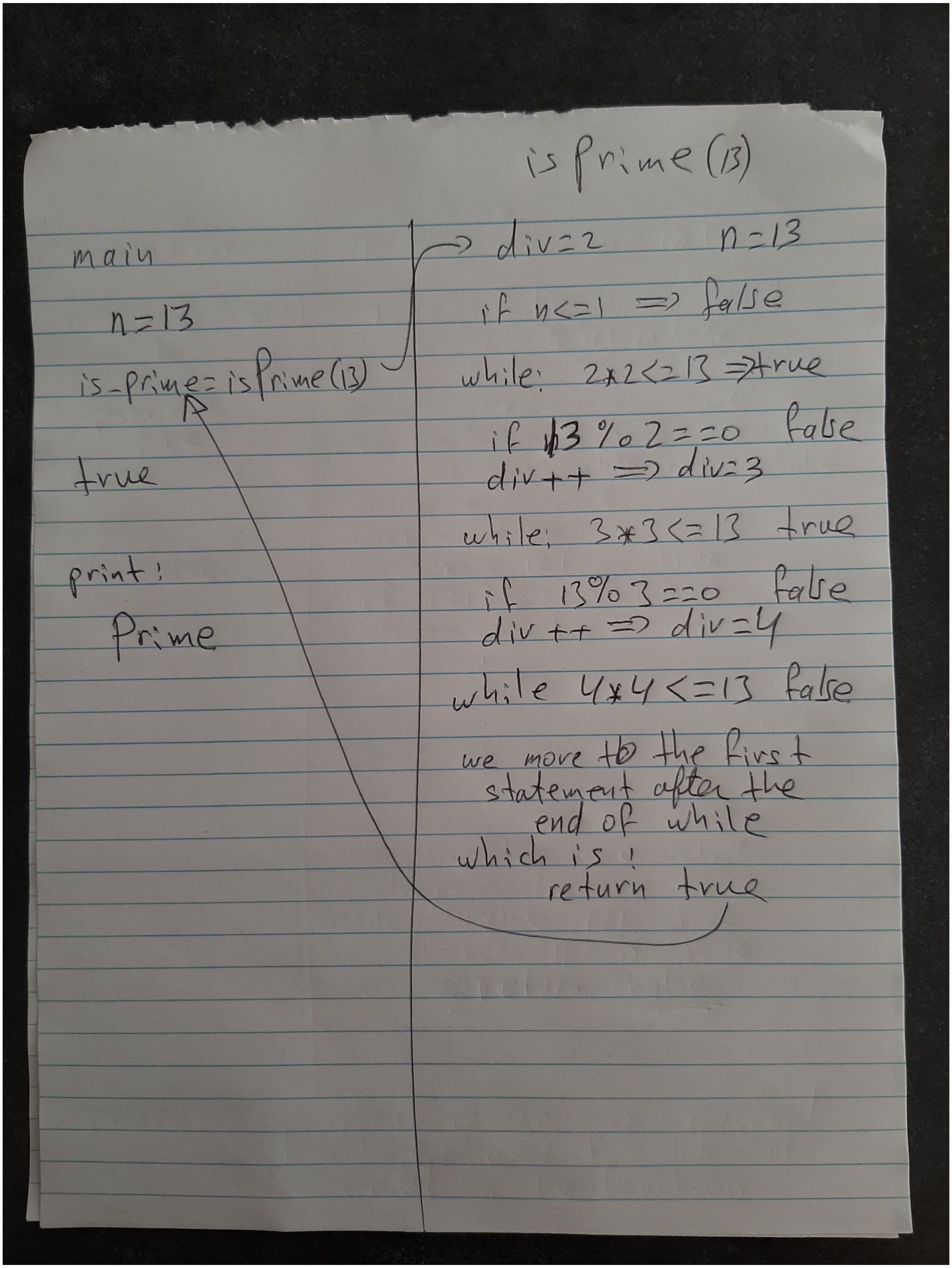
Implement `funumbers.c` in order to be able to run the following program successfully:

```c
#include <stdio.h>
#include <assert.h>
#include "funumbers.h"

int main(void)
{

    assert(is_palindrome(8));
    assert(is_palindrome(111));
    assert(is_palindrome(145541));
    assert(! is_palindrome(14321));

    assert(big_prime(15) == 13);
    assert(big_prime(498) == 491);
    assert(big_prime(3) == 2);

    printf("Good job\n");
    return 0;
 }
```

Tracing:


main
n = 121
is_prime= isPrime(121)
is_prime= false
⇒ print Not Prime
isPrime(121)
div=2
n= 121
121 <=1 false then we don't execute "return false"
while : 2*2 <=121 true
if 121%2==0 false then we don't execute return false
div++ ⇒ div=3
back to while
3*3<=121 is true
if 121%3==0 false
div++ ⇒ div=4
continue the same until
div++ ⇒ div=11
while: 11*11 <= 121 true
if 121% 11==0 true
⇒ return false
remember : when return is reached that will be the last statement to be excuted in the function

i


is Prime(13)
main
n=13
is_prime= isPrime(13)
true
print:
Prime
div=2 n=13
if n<=1 => false
while: 2*2<=13 => true
if 13 % 2 == 0 false
div++ => div=3
while: 3*3<=13 true
if 13%3==0 false
div++ => div=4
while 4*4<=13 false
we move to the first statement after the end of while
which is:
return true

Solutions:

ii

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

int max3(int a, int b, int c)
{
    if (a > b && a > c)
        return a;
    else if (b > a && b > c)
        return b;
    else
        return c;
}

int min3(int a, int b, int c)
{
    if (a < b && a < c)
        return a;
    else if (b < a && b < c)
        return b;
    else
        return c;
}

int middle(int a, int b, int c)
{
    assert(a != b && b != c && a != c);
    return (a + b + c) - max3(a, b, c) - min3(a, b, c);
}

int main(void)
{
    assert(max3(9, 8, 17) == min3(234, 17, 89));
    assert(max3(9, 9, 9) == min3(9, 9, 9));
    assert(max3(19, 9, 19) == min3(99, 19, 19));
    // middle assumes that the three numbers are different
    assert(middle(14, 33, 10) == 14);
    assert(middle(114, 33, 10) == 33);

    printf("Good job\n");
    return 0;
}
```