CS 137

Programming Principles

Chapter 4

Functions, Modules, and Compiling

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 they exist) must have their type declared per variable/parameter. For example, `int fun(int a, int b)` is correct; `int fun(int a, b)` is incorrect.
- The parameters are local variables used only inside the function body.
- `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, use empty brackets: `fun_name();`.

Basic Example

```
#include <stdio.h>
int max(int a, int b)
{
    return a > b ? a : b;
}

int main(void)
{
    printf("%d", max(5, 10));
    return 0;
}
```

Steps:

- Execution always starts with the main function.
- To execute printf on line 9, `max(5, 10)` is called. 5 and 10 are the arguments, matching parameters a and b.
- The value 5 is assigned to a, and the value 10 is assigned to b.
- Since a > b is false (5 > 10 is false), the returned value is b, which is 10.
- This value (10) is returned to the call site, so printf outputs 10.
- Then main returns 0, ending the program.

📌 This function call is by value — 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.

Another Example — swap()

```
#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 main are not the same as the variables x and y in swap. They have the same names, but each has its own space in memory.

Trace table:

x (main)	y (main)	x (swap)	y (swap)	temp (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

x (main)	y (main)	x (swap)	y (swap)	temp (swap)	Output
					bye bye swap
					In main after calling swap:
					x=100, y=200

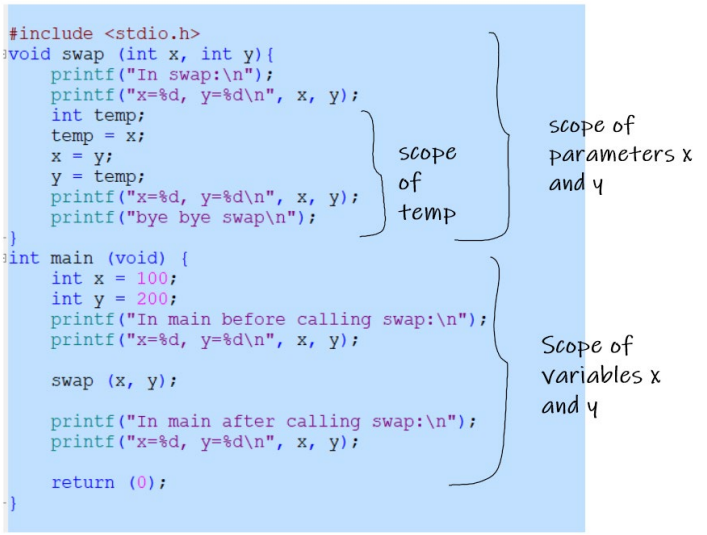
The last statement in swap is printed, then we return to main and can no longer access x and y in swap.

Scope of Variables

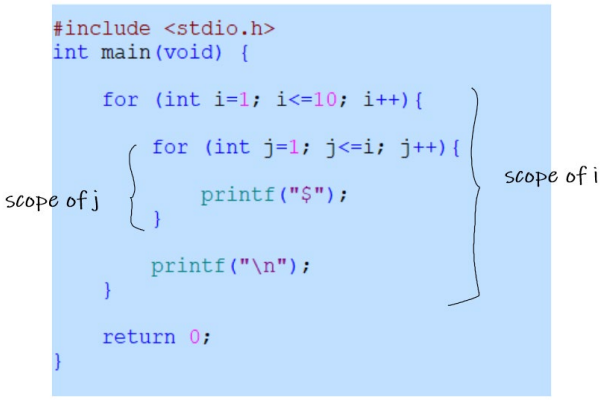
The variables inside the function swap are **local variables**. The values from main are copied into the local variables in the swap function. Changing the local variables inside swap does not affect the variables in main.

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

Example1:



Example2:



Boolean Variables

In C, there are no built-in boolean variables. In C99 and later, the library `<stdbool.h>` provides boolean variables. These `bool` variables are secretly unsigned integers in disguise — they take up a full byte like a char, and can only take the values 0 and 1 (all non-zero values are treated as 1). You can also use the words `true` and `false` with this library.

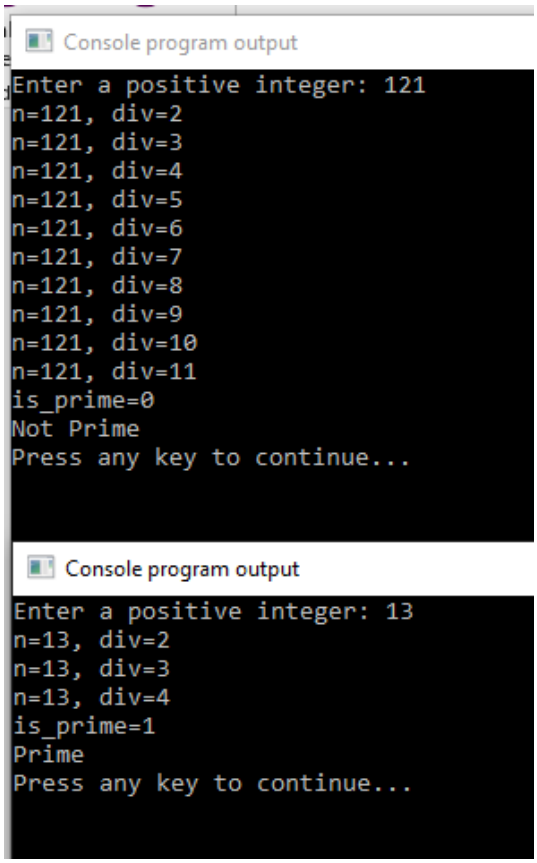
Example

```
#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
    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
        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). The printf statements inside isPrime are included to help trace variable values and understand the process — a useful debugging technique.



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** — a promise to C that we'll eventually define this function with a given return type. The declaration is the first line of the function but ends with a semicolon. The declaration may omit parameter names, though it is advised to include them.

The previous example rewritten with a declaration before main:

```
#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;
    printf("n=%d, div=%d\n", n, div);
    while (div * div <= n)
    {
        if (n % div == 0)
            return false;
        div++;
        printf("n=%d, div=%d\n", n, div);
    }
    return true;
}
```

Some programmers prefer to declare all functions before main and write their implementations after main. Either way, a function must be declared or defined before another function calls it.

assert

Consider the following problem.

The Gregorian calendar replaced the Julian calendar in most of Europe in 1582. North America adopted it in September 1752. A leap year contains an extra day that occurs every four years, except for 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 problems if the year were negative. Since we are in North America, we want the year to be at least 1752. We can accomplish this using assert statements. First add `#include <assert.h>`, then include `assert(year > 1752);` in the function.

```
#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.
- If expr is **false**, it terminates the program with a message showing the filename, line number, function, and expression.

This is great for debugging. It's also good to leave it in (so long as expr is not computationally expensive) because it:

- Helps remember assumptions.
- Causes the program to fail "loudly" rather than "quietly".
- Advises other programmers if the code undergoes modifications.
- Is suitable for regression testing — checking that changes haven't broken anything elsewhere in the code.

Working with Modules and Compiling

In the real world, programs are coded by many programmers. It is often inefficient for them all to work on the same file, and it can get very confusing with 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:

- Readability and maintainability of the program.
- Reusability of the small sub-programs.

Each module has a well-defined **interface** that specifies what services it provides, and an **implementation** part that hides code details from the user (by providing an executable file so they can use the functions listed in the interface without seeing the actual implementation).

Additional advantages:

- Changing the implementation without changing the interface does not require the user to change the main program that uses those modules.
- 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 );
}
```

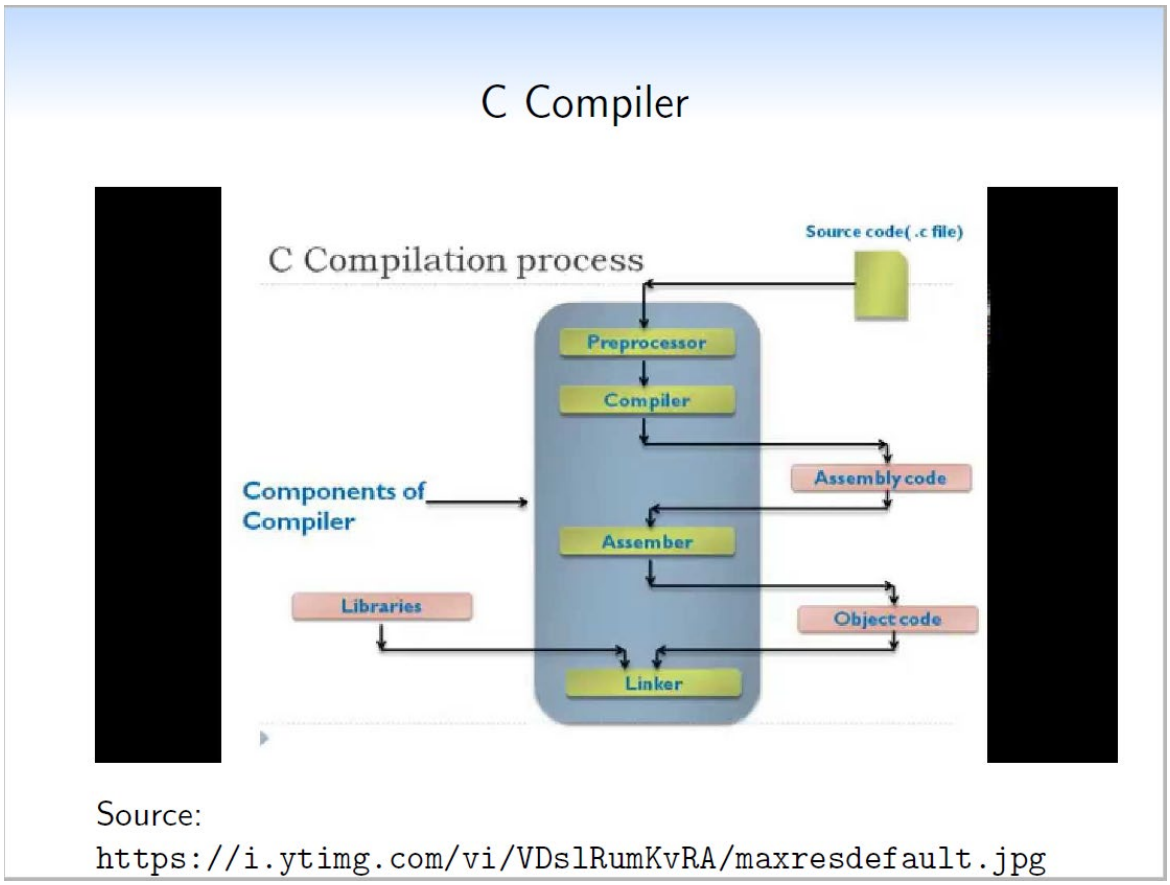
This is the implementation file (.c with the same name as the interface file) which includes the function implementations which were declared in the interface file. It might include helper functions needed for implementation as well (those are not declared in the interface file as users are not supposed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



To compile a C program, we use the `gcc` compiler in Linux (Gnu Compiler Collection). It creates an executable called `a.out` unless you request a different name. When you invoke `gcc`, a series of steps are performed:

The Preprocessor

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

Compiling and Assembling

The compiler translates the C code into assembly language (machine-level code with instructions that manipulate memory and processor directly). The compiler then converts these machine-level instructions into binary code — the **object code**. You can create object code from a C source file with:

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

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

Linking

The linker processes the main function and any possible input arguments, links your program with other programs that contain functions 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: `#include`, `#ifndef`, and `#define`. We can use `#define` to define constants in our code.

Syntax

```
#define identifier replacement
```

Notice there is no equal sign or semicolon — this is a straight text replacement. This is useful for constants in your code. The preprocessor replaces every occurrence of the identifier with its replacement before compilation.

> You can run gcc with the -E flag to see the preprocessor output: $ gcc -E prog.c

Additional Examples

Example 1 — Combinations C(n, r)

Computes the number of combinations of n items taken r at a time.

```
#include <stdio.h>

int factorial(int n);

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,
        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;
    }
    return (product);
}
```

Example 2 — Smallest Prime Divisor

Finds and displays the smallest divisor (other than 1) of integer n, or reports that n is prime.

```
#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 not yet found */

    /* Choose initialization based on whether n is even or odd */
    if (even(n))
        divisor = 2;
    else
    {
        divisor = 0;
        trial = 3;
    }

    /* Test each odd integer as a divisor until one is found
       or until trial > sqrt(n) */
    while (divisor == 0)    {
        if (trial > sqrt(n))
            divisor = n;
        else if ((n % trial) == 0)
            divisor = trial;
        else
            trial += 2;
    }
    return (divisor);
}
int main(void){
    int n, min_div;

    printf("Enter a number between 2 and 1000> ");
    scanf("%d", &n);

    if (n < 2)
        printf("Error: number too small. The smallest prime is 2.\n");
    else if (n <= NMAX){
        min_div = find_div(n);
        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

Try to solve each problem before reviewing the suggested solution.

1.

Complete the following program so that it runs successfully. You may not include any additional interfaces.

```
#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.

Given the following interface file funumbers.h (you may not change it), implement funumbers.c so that the test program below runs successfully.

```
// funumbers.h
#ifndef FUNUMBERS_H
#define FUNUMBERS_H
#include <stdbool.h>

// Pre: num is a valid positive integer
// Returns true if num and its reversed digits are equivalent
// Examples: 12321, 555, 7
// Non-examples: 345, 15
bool is_palindrome(int num);

// Returns the biggest prime number smaller than num
// Pre: num > 2
int big_prime(int num);

#endif
```

```
// main.c — test program
#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;
}
```