Tutorial 2: Structured Programming

The Art of Computer Programming

23 September 2026

Objective. This tutorial puts into practice the constructs of the second lecture: tests, loops, functions, parameter passing, and recursion. Most exercises ask for the same program in Python and in C; write the Python version first, then the C version, and compare them. A few exercises concern only one language, or ask you to predict the behavior of a given program before running it: do make the prediction, it is the whole point.

Parts A to C cover the basics and should be done by everyone; Part D (recursion) is the one to spend the remaining time on. Exercises are roughly ordered by importance within each part; it is fine not to finish them all.

Reminders

- Create a folder `tutorial02` next to `tutorial01` and open it in VS Code. Write each program in its own file, named after the exercise (e.g., `leap.py` and `leap.c`).
- Compile C programs with `$ gcc -Wall -Wextra -std=c23 leap.c -o leap` (or the Run button), and run them with `$ ./leap` (`$ .\leap.exe` on Windows). Take warnings seriously.
- Reading input: `int(input())` in Python; in C, `int n; scanf("%d", &n);`. Reading a floating-point number: `float(input())`; `double x; scanf("%lf", &x);`.
- In Python, `//` is the integer division and `%` the remainder; in C, `/` is the integer division when both operands are integers, and `%` the remainder.

Part A: Tests

Q1. Sign. Write a program that reads an integer and prints whether it is positive, negative, or zero. Use an `if` / `elif` / `else` chain in Python, and `if` / `else if` / `else` in C.

Q2. Leap years. A year is a leap year if it is divisible by 4, except when it is divisible by 100, unless it is also divisible by 400. Write a program that reads a year and prints whether it is a leap year. Write the condition as a single Boolean expression using `and`, `or`, `not` (`&&`, `||`, `!` in C). Test with 1900, 2000, 2024, 2026, and 2100.

Q3. Days of the week. Write a program that reads an integer between 1 and 7 and prints the corresponding day of the week (1 is Monday), or an error message for any other value. Use `match` in Python and `switch` in C.

(a) In the C version, remove the `break` after the case for Wednesday, recompile, and run the program with input 3. Explain.

(b) Modify the program so that it prints "weekday" for 1 to 5 and "weekend" for 6 and 7, with *one* print statement per message: in C, use the fall-through behavior you just observed; in Python, use a case with several alternatives, `case 1 | 2 | 3 | 4 | 5:`.

Q4. Maximum of three. Write a program that reads three integers and prints the largest one, using only tests (no library function). How many comparisons does your program perform in the worst case? Can you do it with two?

Part B: Loops

Q5. Multiplication table. Write a program that reads an integer n and prints its multiplication table from $1 \times n$ to $10 \times n$, one line per product, in the form `3 x 7 = 21`. Use a `for` loop. Then print the whole

table for n from 1 to 10 with two nested loops, one row per n, products separated by spaces (in Python, `print(x, end=" ")` prints `x` without going to the next line; `print()` goes to the next line).

Q6. Sums. Write a program that reads an integer $n \geq 0$ and prints the sum $1 + 2 + \cdots + n$ computed with a loop, next to the value of the formula $n(n+1)/2$. Do it once with a `for` loop, once with a `while` loop. Then also print the sum of the squares $1^2 + \cdots + n^2$ and compare it with $n(n+1)(2n+1)/6$.

C only: what do you get for $n = 100\,000$? Explain using the first lecture, then fix the program with `long long`.

Q7. Fizz buzz. Write a program that prints the integers from 1 to 100, one per line, except that multiples of 3 are replaced by `Fizz`, multiples of 5 by `Buzz`, and multiples of both by `FizzBuzz`. Think about the order of the tests.

Q8. Digits. Write a program that reads a nonnegative integer and prints its number of digits and the sum of its digits, using a `while` loop and only integer division and remainder (no conversion to a string). Check with 0, 7, 1024, and 999 999 999.

Q9. Guess the number. Write a program that chooses a secret integer between 1 and 100, then repeatedly asks the user for a guess, answering "too small", "too large", or "correct" followed by the number of attempts, until the guess is correct.

- *Python:* `import random` then `random.randint(1, 100)`; loop with `while True:` and `break`.
- *C:* `#include <stdlib.h>` and `#include <time.h>`, then `srand(time(nullptr));` once at the start of the program and `rand() % 100 + 1` to draw the number; use a `do ... while` loop.

What is the best strategy for the player? How many attempts does it need at most?

Q10. Primes. Write a program that reads an integer $n \geq 2$ and prints whether it is prime, by testing whether some $d \in \{2, \ldots, n-1\}$ divides n; stop the loop with `break` as soon as a divisor is found. Then:

(a) improve the loop so that it only tests d with $d \times d \leq n$ (why is this enough?);
(b) print all prime numbers up to 1000 with a second, outer loop.

Q11. Collatz. Starting from an integer $n \geq 1$, repeat: if n is even, replace it by $n/2$, otherwise by $3n+1$; stop when $n = 1$. Write a program that reads n and prints the number of steps until 1 is reached (0 for $n = 1$, 1 for $n = 2$, 7 for $n = 3$). Which $n \leq 10\,000$ needs the most steps?

Part C: Functions

Q12. Functions. Turn the code of Q2 and Q10 into functions `is_leap(year)` and `is_prime(n)` returning a Boolean (`bool` in C, with `true` and `false`), and rewrite the programs to use them. A function can have several `return` statements: returning from inside a loop is a natural way to stop it. Then write a function `count_primes(n)` returning the number of primes up to n, and print `count_primes(1000)`.

Q13. Predict before running. Here are two programs. Write down what each prints, then run them to check.

C

```c
#include <stdio.h>

void g(int x) {
  x = x + 1;
  printf("g: %d\n", x);
}

int main() {
  int x = 1;
  {
    int x = 2;
    printf("block: %d\n", x);
  }
  printf("main: %d\n", x);
  g(x);
  printf("main: %d\n", x);
  return 0;
}
```

Python

```python
def g(x):
    x = x + 1
    print("g:", x)


x = 1
if True:
    x = 2
    print("block:", x)
print("main:", x)
g(x)
print("main:", x)
```

Explain each difference using the notions of scope and of parameter passing. What happens in C if you use `x` after the closing brace of a block in which it was declared? And in Python, if `g` prints the variable `y` of the main program, defined before the call, without assigning it? (Assigning a variable inside a function always creates a local variable.)

Q14. Swap. We want a function that exchanges the values of two variables.

(a) In C, write `void swap(int a, int b)` using a temporary variable, call it as `swap(x, y)` and print `x` and `y` afterwards. Why does it not work?
(b) Write `void swap(int *a, int *b)` instead, called as `swap(&x, &y)`.
(c) In C++, write `void swap(int &a, int &b)`, called as `swap(x, y)`. Compare the three versions.
(d) In Python, a function cannot modify the variables of its caller by assignment; the usual idiom is to *return* the two values, `return b, a`, and to assign them on the calling side, `x, y = swap(x, y)`. Try it. (Python programmers write `x, y = y, x` directly.)

Q15. Greatest common divisor. Euclid's algorithm computes $\gcd(a, b)$ for $a, b \geq 0$: while $b \neq 0$, replace (a, b) by $(b, a \bmod b)$; the result is a. Write a function `gcd(a, b)` (in Python and in C), then a function `lcm(a, b)` computing the least common multiple using `gcd`. Check that $\gcd(48, 36) = 12$ and $\mathrm{lcm}(4, 6) = 12$.

Part D: Recursion

Q16. Factorial. Type in the recursive factorial function of the lecture, in Python and in C, and print $n!$ for n from 1 to 25.

(a) Compare the outputs of the two languages. From which n does the C version go wrong, and why? What does `long long` change?
(b) Add a print at the beginning of the function displaying the value of `n`, call `fact(5)`, and explain the order in which the values appear.

Q17. Binary representation. Write a recursive function `print_binary(n)` that prints the binary representation of an integer $n \geq 0$, based on the observation that the binary representation of n is that of $n//2$ followed by the digit $n \bmod 2$. Check with 42 (101010) and 255. What is the base case? In Python, compare with `f"{n:b}"`.

Q18. Fibonacci. The Fibonacci numbers are defined by $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$.

(a) Write the recursive function that follows directly from this definition, and print F_n for n from 0 to 15.
(b) Measure the time taken to compute F_{30}, then F_{35} (Python: `import time` and `time.perf_counter()` before and after the call, the difference is in seconds; C: `#include <time.h>`, `clock()` before and after, the difference divided by `CLOCKS_PER_SEC` is in seconds). How does the time evolve when n increases by 5? Explain by counting how many times `fib(1)` is called.
(c) Write an iterative version with a loop and two variables holding the last two values, and compute F_{90}. (Why not F_{100} in C?)

Q19. Fast exponentiation. Write a recursive function `power(x, n)` computing x^n for an integer $n \geq 0$ using the following: $x^0 = 1$; if n is even, $x^n = (x^{n/2})^2$; if n is odd, $x^n = x \times x^{n-1}$. Check it against a naive loop performing n multiplications. Then write a second recursive function `multiplications(n)` that follows the same case analysis but returns the number of multiplications `power` performs; compare its value with n for $n = 1000$ and $n = 1\,000\,000$. What is the largest power of 2 that fits in a C `long long`?

Q20. How deep can you go? Write a recursive function `count(n)` that returns 0 if $n = 0$ and $1 + count(n-1)$ otherwise (so it just returns n, in the least efficient way). Call it with $n = 100$, 1 000, 10 000, 100 000, 1 000 000 in Python and in C. What happens, and why? Relate it to the call stack described in the lecture. Then write the loop that does the same thing without recursion.

Q21. Towers of Hanoi. Three pegs, n disks of distinct sizes stacked on the first peg from the largest at the bottom to the smallest at the top. Move the whole stack to the third peg, one disk at a time, never putting a disk on a smaller one. Write a recursive function `hanoi(n, source, target, spare)` that prints the sequence of moves ("move disk 1 from A to C"): to move n disks from *source* to *target*, move $n-1$ disks from *source* to *spare*, move the largest disk, then move the $n-1$ disks from *spare* to *target*. How many moves are printed for $n = 3$? For $n = 10$? Find the formula.

Check

Whether or not you reach the end of the sheet, by the end of the session you should be comfortable with: writing a test with several branches; choosing between a `for` and a `while` loop; stopping a loop with `break`; defining a function and returning a value from it; passing a variable by value versus by address in C; writing a recursive function with a proper base case. These are the tools you will use in every program from now on; if one of them is still unclear, ask us, or finish the corresponding exercises at home.