Dirk Praetorius
Michele Ruggeri

Summer semester 2019
April 12, 2019

Scientific programming in mathematics

Exercise sheet 6

Function pointers, strings, structures

Exercise 6.1. A well-known root-finding algorithm is the *secant method*. Let $f : [a, b] \to \mathbb{R}$. Given two initial guesses x_0 and x_1, the algorithm defines the sequence of approximations $(x_n)_{n\in\mathbb{N}_0}$ as

$$x_n := x_{n-1} - f(x_{n-1})\,\frac{x_{n-2} - x_{n-1}}{f(x_{n-2}) - f(x_{n-1})} \quad \text{for } n \geq 2,$$

i.e., the approximation x_n is the root of the line that connects the points $(x_{n-2}, f(x_{n-2}))$ and $(x_{n-1}, f(x_{n-1}))$. Write a function `double secant(double (*f)(double), double x0, double x1, double tau)`, which performs the above iteration until either

$$|f(x_n) - f(x_{n-1})| \leq \tau$$

or

$$|f(x_n)| \leq \tau \quad \text{and} \quad |x_n - x_{n-1}| \leq \begin{cases} \tau & \text{for } |x_n| \leq \tau, \\ \tau|x_n| & \text{else.} \end{cases}$$

In the first case, print a warning to inform the user that the result is presumably wrong. The function returns x_n as an approximation of a zero of f. Use `assert` to check that $\tau > 0$. The function requires as input a suitable implementation `double f(double x)` of the object function f. Moreover, write a main program that reads x_0, x_1, and τ from the keyboard, calls the function, and prints to screen the approximate zero x_n and the function value $f(x_n)$. How can you test your code? What are good examples? Save your source code as `secant.c` into the directory `series06`.

Exercise 6.2. A well-known root-finding algorithm is the *Newton method*. Let $f : [a, b] \to \mathbb{R}$. Given an initial guess x_0, the algorithm defines the sequence of approximations $(x_n)_{n\in\mathbb{N}_0}$ as

$$x_n = x_{n-1} - f(x_{n-1})/f'(x_{n-1}) \quad \text{for } n \geq 1,$$

i.e., the approximation x_n is the zero of the line tangent to the graph of f at the point $(x_{n-1}, f(x_{n-1}))$. Write a function `double newton(double (*f)(double), double (*fprime)(double), double x0, double tau)`, which performs the above iteration until either

$$|f'(x_n)| \leq \tau$$

or

$$|f(x_n)| \leq \tau \quad \text{and} \quad |x_n - x_{n-1}| \leq \begin{cases} \tau & \text{for } |x_n| \leq \tau, \\ \tau|x_n| & \text{else.} \end{cases}$$

In the first case, print a warning to inform the user that the result is presumably wrong. The function returns x_n as an approximation of a zero of f. Use `assert` to check that $\tau > 0$. The function requires as input a suitable implementation `double f(double x)` of the object function f and its derivative `double fprime(double x)`. Moreover, write a main program that reads x_0 and τ from the keyboard, calls the function, and prints to screen the approximate zero x_n and the function value $f(x_n)$. How can you test your code? What are good examples? Save your source code as `newton.c` into the directory `series06`.

Exercise 6.3. Write a function `int checkOccurrence(char* string, char character)`, which, given a string s and a character b, returns how many times b occurs in s. Both the lowercase and the uppercase versions of b contribute to the number of occurrences. Then, write a main program which reads s and b from the keyboard, calls the function, and prints its result to the screen. Test your program appropriately! Save your source code as `checkoccurrence.c` into the directory `series06`.

Exercise 6.4. Write a structure `Date` for the storage of all dates since January 1, 1900 (01.01.1900). The structure consists of three members (day, month, and year) of type `int`. Write the functions

- `Date* newDate(int d, int m, int y)`,
- `Date* delDate(Date* date)`,

as well as the mutator functions

- `void setDateDay(Date* date, int d)`,
- `void setDateMonth(Date* date, int m)`,
- `void setDateYear(Date* date, int y)`,
- `int getDateDay(Date* date)`,
- `int getDateMonth(Date* date)`,
- `int getDateYear(Date* date)`.

Moreover, implement the function `int isMeaningful(Date* date)`, which determines whether a given date is admissible. The function returns the value `1` if the date is admissible, the value `0` otherwise. For instance, the date 31.02.2013 is not admissible. Do not forget to consider leap years! Finally, write a main program to test your implementation in an appropriate way. Save the source code, split into a header file `datum.h` and `datum.c`, into the directory `series06`.

Exercise 6.5. Write a structure `Person` for the storage of sensitive personal information. The structure consists of four members: `firstname` (`char*`), `surname` (`char*`), `address` (`Address*`), and `birthday` (`Date*`). Implement also all necessary mutator functions to work with the structure. Use the structure `Date` from Exercise 6.4 and the structure `Address` from the lecture notes (slide 180). Write the function `Person* whoIsOlder(Person* a, Person* b)`, which compares the age of two persons and returns the oldest. Test your implementation accurately! Save the source code, split into a header file `person.h` and `person.c`, into the directory `series06`.

Exercise 6.6. The *Sieve of Eratosthenes* is an algorithm for the computation of all prime numbers that are smaller than or equal to a given natural number $n_{max} \in \mathbb{N}$. The algorithm consists of the following steps:

- Initialize a list of natural numbers $(2, \ldots, n_{max}) \in \mathbb{N}^{n_{max}-1}$.
- Sweep the list of all multiples of the lowest entry (expect the lowest entry itself).
- Consider the subsequent entry (if any) and sweep all its multiples from the list. Repeat until you reach the final element of the list.

Write a structure `Eratosthenes` for the storage of the result of the algorithm. The structure consists of the upper bound n_{max} (`int`), the number of prime numbers $n \leq n_{max} - 1$ (`int`), and the vector of $\mathbb{N}^n$ of the prime numbers (`int*`). Implement also all necessary mutator functions to work with the structure. Moreover, write a function `Eratosthenes* = doEratosthenesSieve(int nmax)`, which realizes the Sieve of Eratosthenes and uses the structure `Eratosthenes` to return the result. Realize the sweeping in a suitable and efficient way. Note that the vector containing the prime numbers must be of minimal length. Test your implementation accurately! Save the source code, split into `eratosthenes.h` and `eratosthenes.c`, in the directory `series06`.

Exercise 6.7. The prime factorization of a natural number $N \in \mathbb{N}$ is the representation of N as a product of prime numbers, which are usually referred to as prime factors. For example, for $N = 180$, one obtains the factorization $180 = 2^2 \cdot 3^2 \cdot 5$. Write a structure `Factorization` for the storage of such prime factorizations. The structure consists of the number n of prime factors in the decomposition (`int`), the vector in $\mathbb{N}^n$ with the prime factors (`int*`), and the vector in $\mathbb{N}^n$ with the corresponding multiplicities (`int*`). For the previous example ($N = 180$), the structure would contain the members `n=3`, `prime=(2,3,5)` and `multiplicity=(2,2,1)`. Implement also all necessary mutator functions to work with the structure. Write a function

```
Factorization* = computePrimeFactorization(int N),
```

which computes and returns the prime factorization of a given natural number. To obtain a list of all possible prime factors, use the Sieve of Eratosthenes from Exercise 6.6. Moreover, write a function

```
int = recomposeInteger(Factorization* factorization),
```

which performs the opposite process. Test carefully your implementation! Save the source code, split into the files `primefactorization.h` and `primefactorization.c`, in the directory `series06`.

Exercise 6.8. Implement two functions:

- `int = gcd(int a, int b)`, which computes and returns the greatest common divisor (gcd) of two given numbers $a, b \in \mathbb{N}$;
- `int = lcm(int a, int b)`, which computes and returns the least common multiple (lcm) of two given numbers $a, b \in \mathbb{N}$.

Both the gcd and the lcm can be determined by exploiting the prime factorization of the input numbers. In particular, your implementation should exploit the structures and the features of Exercise 6.6 and Exercise 6.7. Save your source code as `gcdlcm.c` into the directory `series06`.