1 An introduction to C++

Exercise 1.1. The sum of two integers

a) Write a program that adds two integers a and b and writes out their sum.

b) Modify the program from a) so that it prompts the user for the values of the two integers.

Filename: `sum_of_two.cpp`

Exercise 1.2. Cooking an egg

The following expression gives the time it takes (in seconds) for the center of the yolk to reach the temperature T_y (in Celsius degrees):

$$t = \frac{M^{2/3} c \rho^{1/3}}{K \pi^2 (4\pi/3)^{2/3}} \ln \left[0.76 \frac{T_o - T_w}{T_y - T_w} \right].$$

Here M, ρ, c and K are the mass, density, specific heat capacity and thermal conductivity of the egg respectively. Relevant values are $M = 47$g for a small egg and $M = 67$g for a large egg, $\rho = 1.038 \mathrm{gcm}^{-3}$, $c = 3.7 \mathrm{Jg}^{-1}\mathrm{K}^{-1}$, and $K = 5.4 \cdot 10^{-3} \mathrm{Wcm}^{-1}\mathrm{K}^{-1}$. Furthermore, T_w is the temperature of the boiling water, and T_o is the original temperature of the egg before being put in the water.

For a hard boiled egg, the center of the yolk should reach $T_y = 70$ C. Make a program that calculates the time it takes to make a hard boiled egg when $T_w = 100$C and the egg is coming directly from the fridge ($T_o = 4$C).

Filename: `egg.cpp`

Exercise 1.3. Sum of first n integers

a) Write a program that computes the sum of the integers from 1 up to (and including) n. Compare with the value of $n(n+1)/2$.

b) Modify the program so that n is asked for in the terminal.

Filename: `integers.cpp`

Exercise 1.4. Generate an approximate Fahrenheit-Celsius conversion table

The formula for converting F degrees Fahrenheit to C degrees Celsius is $C = 5/9(F - 32)$. This can be approximated by $C \approx \hat{C} = (F - 30)/2$.

Write a program that prints a nicely formatted table with three columns: F, C and the approximate value $\hat{C}$. Also write the table to a file.

Filename: `f2c_approx.cpp`

Exercise 1.5. The sum of n integers

Write a program that asks the user how many numbers he wants to add and asks for the value of these numbers. Then the program should print the sum of the numbers.

Filename: `sum_many.cpp`

Exercise 1.6. The sum of n integers from command line

Write a program that prints the sum of the command line arguments.

Hint. For converting from `char` to `int` you can use the function `atoi`.

Filename: `sum_command.cpp`

Exercise 1.7. Making a function in C++

Make a function called `half` that takes an integer argument. The function must print the number it received to the screen, then the program should divide that number by two to make a new number. If the new number is not zero the function then calls the function `half` passing it the new number as its argument. If the number is zero then the function exits.

Call the function `half` with an argument of 100, the screen output should be

```
100
50
25
...
...
1
```

Filename: `half.cpp`

Exercise 1.8. Making an array

Make an array with N uniformly spaced values between a and b. Begin with declaring the array, then fill it with a `for` loop.

Print out the elements of the array to check that the result is as wanted.

Filename: `array.cpp`

Exercise 1.9. Cooking more eggs

a) Modify your program from Exercise 1.2 so that you get a function returning the time it takes for the center of the yolk to reach a temperature T_y when the egg had a temperature T_o before cooking. Check that you get the same result as in Exercise 1.2 for $T_o = 4$ C and $T_y = 70$ C.

b) Make an array of T_y values, $T_y = \{60, 62, 64, 66, 68, 70.72\}$. Then declare an array `t` of the same length for time values. Make a `for` loop and fill in the `t` array.

c) Use a `for` loop to print out a nicely formatted table of T_y values and the corresponding t values.

Filename: `eggs.cpp`

Exercise 1.10. Stirling's approximation

Stirling's approximation is

$$\ln x! \approx x \ln x - x.$$

a) Write a function taking an integer value x as argument that returns Stirling's approximation to $\ln x!$.

b Print a nicely formatted table with three columns: x, $\ln x!$ and Stirling's approximation to $\ln x!$ for $x = 2, 5, 10, 50, 100, 1000$. Also write this table to a file named `stirling.txt`.

Hint. To compute $\ln x!$ you can use `lgamma` from `<cmath>` Then $\ln x! =$ `lgamma(x+1)`.

c) Make sure you have stored the x values in an array `x`. Then declare two arrays `exact` and `approx` of the same length as `x`. Use a `for` loop to fill these arrays with the exact value and the approximated value of $\ln x!$ for each value in the array `x`.

Filename: `stirling.cpp`

Exercise 1.11. Primality checker

Recall that a prime number is a number greater than 1 that has exactly 2 divisors. Said differently, a number greater than one is a prime if it is divisible by only itself and one. Every number n can be written as a unique product of primes (e.g. $12 = 2 \cdot 2 \cdot 3$), this is called the prime factorization of n.

Make a function that takes a number n, and returns true if it's prime, and false if it's not. Use the program to find all prime numbers up to 100.

Hint. You will only need to check divisibility for numbers up to and including $\sqrt{n}$, because any greater divisor will imply that there is a divisor less than this.

Filename: `prime.cpp`

Exercise 1.12. Eulers totient function

Two numbers n and m are called relatively prime if they have no common divisors except for 1. That is, no number greater than one should divide both numbers with no residue.

a) Make a function that takes two numbers and returns true if they're relatively prime and false if they're not.

b) Euler's totient function is defined as

$$\phi(d) = \#\{\text{Numbers less than d which are relatively prime to d}\}.$$

Implement Eulers totient function and print $\phi(d)$ for $d = 10, 50, 100, 200$.

Filename: `euler.cpp`

Exercise 1.13. Converting from base n to decimal

Make a function `long convert_n(long number, int n)` that converts a number from base n to decimal for $n = 2, 3, 4, 5, 6, 7, 8, 9, 10$.

Hint. Remember that a number $d_0 d_1 d_2 ... d_k$ in base n is the following in decimal: $d_0 \cdot n^k + d_1 \cdot n^{k-1} + d_2 \cdot n^{k-2} + ... + d_k \cdot n^0$. Some useful operations: `/` and `%`.

Filename: `convert_n.cpp`

Exercise 1.14. Converting from hexadecimal to decimal

Make a function `long convert_hex(string number)` that converts a number of the type string from hexadecimal to decimal. The answer should be returned as type long.

Hint. When converting from `string` to `int` or `long`, subtract the zero string $'0'$ before converting or use the function `atoi`.

Filename: `convert_hex.cpp`

Exercise 1.15. Add two binary numbers

a) Make a function `long add_binary(long a, long b)` that adds two binary numbers a and b.

Hint. Add like you do by hand: start with the last digits. You can use a `while` loop containing

```
sum += long((a%10 + b%10 + r)%2)*pow(10, i++);
r = int((a%10 + b%10 + r)/2);
a /= 10;
b /= 10;
```

Think about what `r` is. What should you do if `r` is non-zero after you have gone through all the digits?

b) Modify your program from a) so that you can add any two numbers of base n for $n = 2, 3, 4, 5, 6, 7, 8, 9, 10$.

Filename: `add_n.cpp`

Exercise 1.16. Adding fractions

Write a program to add two fractions and display the resulting fraction. Your program will prompt the user to input the two fractions. The numerator and denominator of each fraction are input separately by space, as illustrated below.

Enter fraction 1 (numerator denominator): 1 3

Enter fraction 2 (numerator denominator): 2 5

Result: 11/15

Hint. You will need to use a struct to define a fraction. The struct has two members: numerator and denominator.

Filename: `fraction.cpp`

Exercise 1.17. Adding and simplifying fractions

Modify your program from Exercise 1.16 so that the result is the fraction in it's simplest form. You should make a function `simplify` that takes in a fraction and simplifies it. Let the function be of the type `void`.

An example from the terminal:

Enter fraction 1 (numerator denominator): 1 3

Enter fraction 2 (numerator denominator): 2 6

Result: 2/3

Hint. For the function to be able to change the value of the fraction, the argument of the function should be a reference variable: `simplify(Fraction& fract)` where `Fraction` is the name of the struct for fractions.

Filename: `fraction2.cpp`

Exercise 1.18. Make a class for rectangles

Make a class `Rectangle` that has two private variables and one member function which will return the area of the rectangle.

Filename: `Rectangle.cpp`

Exercise 1.19. Make a class for cooking eggs

a) Make a class `Cook_egg` that has a public function taking no arguments which returns the time it takes for the egg to be cooked.

b) Add two different methods for changing the mass of the egg. The first method takes the mass of the egg in grams as argument and reassigns the mass of the egg to this mass. The other method takes 'S', 'M' or 'L' as arguments and changes the mass to 47g, 57g or 67g respectively.

c) Add two new methods: one for changing the initial temperature of the egg and one for changing the desired final temperature of the yolk.

Filename: `Cook_egg.cpp`

Exercise 1.20. Make a class for quadratic functions

Consider a quadratic function $f(x; a, b, c) = ax^2 + bx + c$. Make a class `Quadratic` for representing f, where `a`, `b` and `c` are initial arguments. The class should have three methods: `value`, `table` and `roots`.

The `value` method should compute the value of f at a point x. The `table` method should write out a table of x and f values for n uniformly spaced x values in the interval $[L, R]$. The `roots` method should compute the two roots of the quadratic function. It should accept complex roots.

Hint. For the `roots` method to be able to return two values, consider making a structure `Two_vals` containing two values.

Filename: `Quadratic.cpp`

Exercise 1.21. Points in different coordinate systems

Make a class `Point` to represent a point (x, y, z) in space. This class should have methods for getting the value of x, y and z.

Make a subclass `SphericalPoint` that inherits from `Point`. The subclass should take the spherical representation of a point (r, ϕ, θ) as arguments. Call the superclass constructor with the corresponding x, y and z values (recall that $x = r\cos\phi\sin\theta$, $y = r\sin\phi\sin\theta$ and $z = r\cos\theta$).

Verify the implementation by initializing three points (e.g. the three Cartesian unit vectors) as spherical points and print the corresponding Cartesian coordinates by calling the methods for getting the value of x, y and z.

Filename: `Point.cpp`

Exercise 1.22. Numerical approximations for the derivative

Let $f(x)$ be a function and $f'(x)$ its derivative. There are many ways to approximate the derivative, some of which are:

$$
\begin{aligned}
f'(x) &\approx \frac{f(x+h) - f(x)}{h}, \\
f'(x) &\approx \frac{f(x+h) - f(x-h)}{2h}, \\
f'(x) &\approx \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}.
\end{aligned}
$$

Make a class `Diff` with a function f as initial argument and implement three methods `diff1`, `diff2`, and `diff3` for approximating the derivative using the above formulas. The class should also have a method `set_h` for changing the value of h.

Let $f(x) = e^x$ and compute $f'(1)$ with the three different methods for $h \in \{1, 0.5, 0.2, 0.1, 0.01, 0.001\}$. You should let h be an array with the different values and loop over it.

Hint. To send a function as an argument to a class you should use pointers, i.e. let the beginning of the class be

```
class Diff
{
private:
  double (*f)(double x);
  double h;

public:
  Diff(double function(double x), double _h = 0.001)
  {
    f = function;
    h = _h;
  }
```

Filename: `Diff.cpp`

Exercise 1.23. Numerical approximations for integration

Let $f(x)$ be a function we want to integrate. The integral $\int_a^b f(x)dx$ can be approximated in many different ways, some of which are: the midpoint rule, the trapezoidal rule and Simpson's rule.

The midpoint rule gives the following approximation to the integral:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{N-1} f(a + \Delta x(i + 1/2)) \cdot \Delta x,$$

where $\Delta x = (b - a)/(N)$ and N is the number of intervals the integral is divided into.

The trapezoidal rule gives the following approximation to the integral:

$$\int_a^b f(x)dx \approx \frac{1}{2}\left(f(a) + 2\sum_{i=1}^{N-1} f(a + \Delta x \cdot i) + f(b)\right)\Delta x.$$

Simpson's rule gives the following approximation to the integral:

$$\int_a^b f(x)dx \approx \frac{1}{3}\left(f(a) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + ... + 4f(x_{N-1}) + f(b)\right)\Delta x,$$

where $x_i = a + \Delta x \cdot i$.

Make a class `Integration` where you implement the three different rules. Include also a method for changing the number of intervals N.

Filename: `Integration.cpp`

Exercise 1.24. Newton's method

Make a class `Function` which is a subclass of `Diff` from Exercise 1.22. It should take a function $f(x)$ as an initial variable.

Make a method `call` that takes x as an argument and returns the value of the function for that x.

We would like the class to give estimated values for roots of f. That is, points such that $f(x) = 0$. To do this we implement Newton's formula. It is given recursively as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)},$$

where we give a starting point x_0. In some cases (not all) x_n will approach a root of f. Implement this in a method `approx_root` that takes a starting point and a bound $\epsilon < 1$ as arguments and approximates x_n such that $f(x_n) < \epsilon$.

Hint. Implement a simple convergence test. Check that $f(x_n) < 1$ after 100 iterations. If not terminate the loop and inform the user that there is no convergence for that starting point. It is still a possibility for convergence, but unlikely.

Filename: `Function.cpp`