The C programming language (exercises)
— P.02 —

To compile C and C++ programs, Windows 10 and Windows 11 users should install the Windows Subsystem for Linux. As an alternative, they can install instead an Integrated Development Environment (IDE), such as Visual Code Studio or Eclipse. GNU/Linux and MacOS users should be fine. Nonetheless, in order to be able to compile 32-bit executables, GNU/Linux users should install the `build-essential` and the `gcc-multilib` packages (these are for the `ubuntu` distribution, for other distribution their names may be different). Study carefully the provided C source code.

Summary:

- How to compile and run a program (GNU/Linux)
- How to manage archives
- The "Hello World" program
- Program to print some numbers
- Program to print the size in bytes of the fundamental data types
- Computation of Fibonacci numbers
- Printing all command line arguments
- Integer arithmetic pitfalls
- A more elaborate example (integer factorization)
- Final example (rational approximation)
- gdb and valgrind
- Homework

How to compile and run a program (GNU/Linux)

A C program is composed by one or more `.c` source files and by zero or more `.h` files (included by the `.c` source files). To compile the program under GNU/Linux, the following command should be used:

```
cc -Wall -O2 source_files... -o executable_name -lm
```

Replace `source_files...` by the list of the `.c` source files, and replace `executable_name` by the name you desire to give to the executable file. If `-o executable_name` is omitted from the command line, the executable will get by default the name `a.out`. The option `-Wall` instructs the compiler (`cc`) to give you a warning whenever you use dubious C code (such as using an uninitialized variable). The option `-O2` instructs the compiler to optimize the program for speed. The linker option `-lm`, that must be placed at the end, instructs the compiler to link the program with the math library (so that functions like `sin()` and the like are properly taken care of). For example, if your program is composed by the files `source1.c`, `source2.c`, if they include the file `source.h`, and if you desire the executable to be named `prog`, the command should be

```
cc -Wall -O2 source1.c source2.c -o prog -lm
```

and you can run it on a terminal using the command

```
./prog
```

You can automate the process of compiling the program using a `makefile`. Put the text (beware of the tab, denoted below by an arrow)

```
prog:   source1.c source2.c source.h
——————→cc -Wall -O2 source1.c source2.c -o prog -lm
```

in a file named either `Makefile` or `makefile`. Running the command

```
make
```

will recompile your program if at least one of the source files has changed since the last compilation.

How to manage archives

All source code files for this class, together with the makefile needed to (re)compile all programs, can be found in the compressed tar archive P02.tgz. On a GNU/Linux system, to extract the files it holds on a terminal, go to the directory (folder) where you want to extract the files (use the cd command to do this), and then use the command

```
tar xzvf P02.tgz
```

to extract the files. They should appear in a new directory named P02. To create an archive use the command

```
tar czvf name_of_your_archive.tgz list_of_files_and_directories_to_put_in_the_archive
```

For more information about a command, just run the man command. For example, to get more information about the tar program, run

```
man tar
```

The “Hello World” program

The hello.c file contains the C code

```
/*
** Hello world program
*/

#include <stdio.h>

int main(void)
{
  puts("Hello world!");
  return 0;
}
```

Compile it and run it. Modify the code to print “Hello X”, where X is your name.

Program to print some numbers

The following program (`table.c`) prints the first ten positive integers, their squares, and their square roots.

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

void do_it(int N)
{
  int i;

  printf(" n n*n      sqrt(n)\n");
  printf("-- --- -----------------\n");
  for(i = 1;i <= N;i++)
    printf("%2d %3d %17.15f\n",i,i * i,sqrt((double)i));
}

int main(void)
{
  do_it(10);
  return 0;
}
```

Compile and run it. Modify the program to print the sine and cosine of the angles $\mathbf{0, 1, 2, \ldots, 90}$ (in degrees) and to place the output in a file named `table.txt`.

Hints:

- Use the help system (on GNU/Linux use the command `man`) to study how the formatting string of the `printf` function is specified. Study also the functions `fopen`, `fclose` and `fprintf`.
- The argument of the `sin` and `cos` functions are in radians; the value of π is given by the symbol `M_PI`, which is defined in `math.h`. To convert from degrees to radians multiply the angle by `M_PI/180.0`.

Program to print the size in bytes of the fundamental data types

The file `sizes.h` contains the code

```
#ifndef SIZES_H
#define SIZES_H
void print_sizes(void);
#endif
```

The file `sizes.c` contains the code

```
#include <stdio.h>
#include "sizes.h"
void print_sizes(void)
{
  printf("sizeof(void *) ...... %d\n",(int)sizeof(void *)); // size of any pointer
  printf("sizeof(void) ........ %d\n",(int)sizeof(void));
  printf("sizeof(char) ........ %d\n",(int)sizeof(char));
  printf("sizeof(short) ....... %d\n",(int)sizeof(short));
  printf("sizeof(int) ......... %d\n",(int)sizeof(int));
  printf("sizeof(long) ........ %d\n",(int)sizeof(long));
  printf("sizeof(long long) ... %d\n",(int)sizeof(long long));
  printf("sizeof(float) ....... %d\n",(int)sizeof(float));
  printf("sizeof(double) ...... %d\n",(int)sizeof(double));
}
```

The file `main.c` contains the code

```
#include "sizes.h"
int main(void)
{
  print_sizes();
  return 0;
}
```

Study the way the program code is split among the three files. Compile and run the program. On a 64-bit machine, add `-m32`, `-mx32`, or `-m64` to the compilation flags and check if any of the sizes reported by the program changes.

Computation of Fibonacci numbers

The program in the file `fibonacci.c` computes Fibonacci numbers using the four different methods briefly described in this slide. Compare the code of this file with the one presented in the slide.

Compile and run the program. Make graphs of the execution times of each of the four functions reported by the program. Explain why the function `F_v1` is extremely slow (hint: compare the execution time of that function with the value it returns). Estimate how long your computer will take to compute $\boldsymbol{F}_{60}$ using the `F_v1` function.

The code in the file `fibonacci_with_a_macro.c` is almost identical to the code in the file `fibonacci.c`. In the original code, inside the `for` loop of the `main` function, there are four lines of code, with several statements, that are almost identical. In the modified code they are replaced by a single macro definition and four macro invocations. This guarantees consistency. Use, for example,

```
vim -d fibonacci.c fibonacci_with_a_macro.c
```

or

```
meld fibonacci.c fibonacci_with_a_macro.c
```

to see the differences between the two files. Note that inside a macro replacement text, a single # stringifies the next token (i.e., converts it into a string), and ## concatenates the tokens on its left and right hand sides into a single token, so that, for example, `dt ## 1` becomes the single token `dt1` (an identifier), and not the two tokens `dt` (an identifier) and `1` (a number).

Printing all command line arguments

The following program (`command_line_arguments.c`) prints all its command line arguments.

```c
#include <stdio.h>

int main(int argc,char *argv[argc])
{
  for(int i = 0;i < argc;i++)
    printf("argv[%2d] = \"%s\"\n",i,argv[i]);
  return 0;
}
```

Study it with care. Modify it to print after each argument the integer it represents (if any). It is possible to convert a string to an integer using the `atoi` function; it does not perform any checks on its input. The more advanced `strtol` can do that, but it is harder to use. Investigate how you can use it.

Integer arithmetic pitfalls

The following program (`integer_arithmetic_pitfalls.c`) contains a small amount of code that performs some simple integer operations.

```c
#include <stdio.h>

int main(void)
{
  unsigned int i = 1;
  int j = -1;
  int k = -2147483648;

  printf("original i = %u\n",i);
  printf("original j = %d\n",j);
  printf("original k = %d\n",k);
  // compare i with j
  if(i > j)
    printf("i > j is true\n");
  else
    printf("i > j is false\n");
  // replace k by its absolute value and print the result
  if(k < 0)
    k = -k;
  printf("absolute value of k = %d\n",k);
  return 0;
}
```

Before compiling and executing the program, in your opinion what should the output of the program be? Now, compile and run it. Were your predictions correct? If so, congratulations. If not, explain why not!

```
           i > j is false
absolute value of k = -2147483648
```

A more elaborate example (integer factorization)

The following program (`factor.c`) computes the factorization of an integer.

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

int factor(int n,int prime_factors[16],int multiplicity[16])
{
  int d,n_factors;

  n_factors = 0;
  for(d = 2;d * d <= n;d = (d + 1) | 1) // d = 2,3,5,7,9,...
    if(n % d == 0)
    {
      prime_factors[n_factors] = d; // d is a prime factor
      multiplicity[n_factors] = 0;
      do
      {
        n /= d;
        multiplicity[n_factors]++;
      }
      while(n % d == 0);
      n_factors++;
    }
  if(n > 1)
  { // the remaining divisor, if any, must be a prime number
    prime_factors[n_factors] = n;
    multiplicity[n_factors++] = 1;
  }
  return n_factors;
}
```

```c
int main(int argc,char *argv[argc])
{
  int i,j,n,nf,f[16],m[16]; // 16 is more than enough...

  for(i = 1;i < argc;i++)
    if((n = atoi(argv[i])) > 1)
    {
      nf = factor(n,f,m);
      printf("%d = ",n);
      for(j = 0;j < nf;j++)
        if(m[j] == 1)
          printf("%s%d",(j == 0) ? "" : "*",f[j]);
        else
          printf("%s%d^%d",(j == 0) ? "" : "*",f[j],m[j]);
      printf("\n");
    }
  return 0;
}
```

Study the program. Compile and run it (for example, `./factor 30`). Why is the program slow when n is a prime number larger than 46339^2, such as 2147483647? (Hint: arithmetic overflow in the signed integer data type.) Get rid of that programming bug.

Homework challenge: Modify the program so that it outputs all possible divisors of `n`. [Hint: there are `(1+m[0])*(1+m[1])*...*(1+m[nf-1])` divisors.]

Final example (rational approximation)

The program in the file `rational_approximation.c` computes the best rational approximation to a given real number using two different approaches. One is based on a slow but straightforward brute force search for the best solution, and the other (fast, but not the fastest possible!) is based on some interesting mathematical properties of best rational approximations.

Study the program. Pay attention to the data types that are defined and how preprocessor directives can enable or disable (at compile time) parts of the program. Try to understand how the (slow) brute force search works. Attempting to understand the mathematics behind the fast method lies outside the scope of AED (for the curious, it uses a mix of a so-called Stern-Brocot tree and of a continued fraction expansion).

Compile the program and run it. Confirm that the two methods give the same result. Experiment with other real numbers. For example, if the line "`x = M_PI;`" is replaced by the line "`x = exp(1.0);`" or by the line "`x = M_E;`" the program computes best rational approximations to e (base of the natural logarithms).

Modify the program to count and print, if `DEBUG` is negative, the number of tests `e < best_e` that are performed by each of the two functions. Do this for several interesting real numbers, such as π, e, $\sqrt{2}$, and, say, $(1+\sqrt{5})/2$. What can you say about the growth of the number of tests as a function of `max_den` for each of the two functions?

Measure approximately the time it takes your computer to compute

```
best_rational_approximation_slow(M_PI,100000000u)
```

and to compute

```
best_rational_approximation_fast(M_PI,100000000u)
```

Try also other values of the second argument and make graphs of the execution time versus this second argument.

gdb and valgrind

Three of the programs stored in the archive `P02.tgz`, namely `binary_search.c`, `count_words.c`, and `primes.c`, contain errors or memory leaks. Correct them.

The `count_words` program requires the name of a text file as argument. In the `P02.tgz` archive one such file is supplied: `SherlockHolmes.txt`. To run the program just do (on a terminal)

```
make count_words
./count_words SherlockHolmes.txt
```

As a bonus, you may also read some of the Sherlock Holmes stories. . .

Homework (optional but recommended, part I)

Problem 1: Write a program that finds all integers n that have the following characteristics:

- the integer has exactly 10 base-10 digits,
- the integer cannot have repeated digits (so all digits must appear exactly once), and
- for $k = 1, 2, \ldots, 10$, the integer formed by the first k **most significant** base-10 digits of n must be divisible by k.

For example, n cannot be 1295073846 because, among other problems, 1295 is not divisible by 4 (but 1 is divisible by 1, 12 is divisible by 2 and 129 is divisible by 3.

How about other bases? On a 64-bit processor with an `unsigned long` data type of 64 bits, it should be possible to go up to base 16.

Problem 2: The triangular numbers are the integers $0, 1, 3, 6, 10, \ldots$. They are given by the formula $k(k+1)/2$ for $k = 0, 1, 2, 3, \ldots$. (It is more usual to exclude the number 0 from this list, but for our purposes it is more convenient to include it.) Write a program that, for N equal to 100, 1000, 10000, 100000, and 1000000, computes how many integers in the interval $[0, N]$ are

- the sum of exactly two triangular numbers;
- the sum of exactly three triangular numbers.

Historical remark: the famous mathematician Carl Friedrich Gauss proved that $n = \triangle + \triangle + \triangle$.

Homework (optional but recommended, part II)

Problem 3 [the minimum overlap problem]: Let n be a positive integer and let $A = \{\, a_1, a_2, \ldots, a_n \,\}$ be a subset of the set $\{\, 1, 2, \ldots, 2n \,\}$ with exactly n elements, and let $B = \{\, b_1, b_2, \ldots, b_n \,\}$ be the complementary set. For example, for $n = 3$, we may have

$$A = \{\, 1, 4, 5 \,\}$$

and

$$B = \{\, 2, 3, 6 \,\}.$$

Let $M_n(A, k)$ be the number of solutions of $a_i - b_j = k$, $1 \leqslant i, j \leqslant n$, and let $M_n(A)$ be the **largest** value of $M_n(A, k)$. For example, for the above example we have 9 differences:

$a_1 - b_1 = -1$	$a_1 - b_2 = -2$	$a_1 - b_3 = -5$
$a_2 - b_1 = +2$	$a_2 - b_2 = +1$	$a_2 - b_3 = -2$
$a_3 - b_1 = +3$	$a_3 - b_2 = +2$	$a_3 - b_3 = -1$

It follows that

k	-5	-2	-1	$+1$	$+2$	$+3$
$M_3(A, k)$	1	2	2	1	2	1

and, finally, that $M_3(A) = 2$. The minimum overlap problem asks, for a given n, what is the **smallest** value of $M_n(A)$, i.e., it asks for the value of $M_n = \min_A M_n(A)$. It is known that

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
M_n	1	1	2	2	3	3	3	4	4	5	5	5	6	6	6

Your task is to confirm these values and to compute at least one more. I know them up to $n = 25$.