电子科技大学
格拉斯哥学院
Glasgow College, UESTC

UESTC(HN) 1005 Introductory Programming Lab Manual

S1 2025–26

Contents

Lab Session I

1.1 Variables

In the C programming language, you must explicitly declare variables and their types. In a sense, variables serve as placeholders in the memory where program data are stored. In computer memory, everything is stored as `0`s and `1`s, i.e., as *bits*, regardless of type. Data types indicate how contiguous sequences of bits should be interpreted and utilised. Variables can be of different types, which will determine how a variable's value is stored in memory as contiguous bits.

1.1.1 Data Type

Each variable in C has an associated data type. It specifies the type of data stored by the variable, e.g., integer, character, string (a sequence of characters), floating point number, etc. Each data type requires different amounts of memory and affects what happens when you invoke a particular operator, e.g., the output of the division operator `/` is different if both arguments are integers or both arguments are floating point numbers. Table 1.1 lists the data types that are typically used in C. You must be careful in selecting the right type of variable, as it can affect the execution of your programs.

Table 1.1: Common C data types for numerics.

Data Type	Size (bytes)[1]	Lowest	Highest	Smallest > 0 (normal)[2]	Format Specifier
`signed char`[3]	1	-2^{7}	$2^{7}-1$	—	`%c` [4]
`int`	4	-2^{31}	$2^{31}-1$	—	`%d`
`long long int`	8	-2^{63}	$2^{63}-1$	—	`%lld`
`float`	4	$\approx -3.40 \times 10^{38}$	$\approx 3.40 \times 10^{38}$	$\approx 1.18 \times 10^{-38}$	`%f`
`double`	8	$\approx -1.80 \times 10^{308}$	$\approx 1.80 \times 10^{308}$	$\approx 2.23 \times 10^{-308}$	`%lf`.

In this lab session, we will focus on two data types, `int` and `float`. As an example, a variable of type `int` can only store integer numbers, i.e., numbers without any decimal points. On the other hand, `float` type variables can store single-precision floating-point numbers that can be much larger than `int`, but have at most 8 significant digits. There are many other data types in the C language, such as `unsigned int`. The difference between `int` and `unsigned int` is that `unsigned int` values must be non-negative.

[1] Typical on 64-bit desktops/laptops and also on the 32-bit STM32. Sizes can vary by platform.

[2] IEEE-754 normal minimum; subnormals exist down to $\approx 1.4 \times 10^{-45}$ for `float` and $\approx 4.94 \times 10^{-324}$ for `double`.

[3] Plain `char` may be signed or unsigned. Use `signed char` / `unsigned char` to be explicit.

[4] Print as a number with `%hhd`; as a character with `%c`.

The `sizeof` operator returns the size, in bytes, of its operand, and the output of `sizeof` is an unsigned value. For example, `sizeof(int)` returns the size of `int`, specifically, 4 B[5]. Additionally, we can obtain the range of different data types using the header `<limits.h>`. For example, the following program will output the size and range of variables of type `int`. Please try to print the size and range of other data types according to this example.

Code 1.1: `byte.c`

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

int main(int argc, char **argv) {
    printf("Storage size for int in bytes: %d \n", sizeof(int));
    printf("Maximum value of int: %d\n", INT_MAX);
    printf("Minimum value of int: %d\n", INT_MIN);
    return 0;
}
```

```
Storage size for int in bytes: 4
2147483647
-2147483648
```

1.1.2 Input and Output

To complete the exercises in this lab, you will need to use the functions `scanf` and `printf` to input and output data, respectively. The following example illustrates how to use these functions.

Code 1.2: `variable.c`

```c
#include <stdio.h>

int main() {
    int some_integer;
    float some_number;
    char some_char;
    scanf("%d %f %c", &some_integer, &some_number, &some_char);
    printf("The input data are %d, %f, %c", some_integer, some_number,
        some_char);
    return 0;
}
```

[5] See footnote 1.

1.2 Conditional Statement (Branch Structure)

A conditional statement in C is designated by one of two keywords: `if` or `switch`. The semantics of an `if` statement is fairly simple. If the statement is true, the code block (indicated by a statement or the contents of a compound statement, i.e., `{...}`, block) is run. A closely related conditional statement is `if ... else`, which executes the `if` code block when the condition is true and executes the `else` code block statement when false. A complete example of an `if ... else` conditional statement is shown in Code 1.3.

Code 1.3: `leapYearIf.c`

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int year;
    scanf("%d", &year);
    if(year % 100 == 0) {
        if(year % 400 == 0)
            printf("%d is a leap year.", year);
        else
            printf("%d is a nonleap year.", year);
    }
    else if(year % 4 == 0)
        printf("%d is a leap year.", year);
    else
        printf("%d is a nonleap year.", year);
    return 0;
}
```

Pay attention to the following when using `if ... else`:

1. In C, `x = 1` and `x == 1` do completely different things. The single equal operator `=` indicates assignment: it assigns the value of `1` to the variable with identifier `x`. The double equal operator `==` indicates comparison: it returns true if x is equal to 1 and `0` otherwise.
2. If you forget to use the `{...}` code block, it may be very easy to make later mistakes. The computer will happily execute whatever instructions you write regardless of your actual intent. Specifically in this case, the compiler will only consider the closest statement (all text up to the closing semicolon) as the `if` condition. This is illustrated in Code 1.4.

Code 1.4: `noBracket.c`

```c
#include <stdio.h>

int main(int argc, char **argv) {
    int x = 0;
    if(x == 1)
```

```c
        printf("x is 1.\n");
        printf("This will always be printed!");
    return 0;
}
```

```
This will always be printed!
```

1.3 Functions

You are familiar with functions in mathematics; for example, you have encountered functions such as $g(x) = x^2 + 2x - 3$. In this case, the function *returns* a value. The returned value is computed based on the input. You have also seen functions with multiple arguments or inputs: $h(x, y) = 5x - 6x + x * y$.

We use functions like this in programming. In C, we need to specify the type of variable being returned, e.g., `int` or `float`. We also need to specify the type of each input variable. We also sometimes use functions which do not return any values — these are useful if we want to repeat commands or improve program organization. These functions are called `void` functions or procedures.

Functions can make programs more readable. Consider Code 1.5 which encapsulates the leap year detection portion of Code 1.3 in the function `is_leap_year`. Please note that there is no built-in `true` value in C, so we use the standard convention that `1` and `0` represents true and false, respectively.

Code 1.5: `leapYearFunction.c`

```c
#include <stdio.h>

int is_leap_year(int year) {
    if(year % 100 == 0) {
        if(year % 400 == 0)
            return 1;
        else
            return 0;
    }
    else if(year % 4 == 0)
        return 1;
    else
        return 0;
}

int main(int argc, char **argv) {
    int year;
    scanf("%d", &year);
    if(is_leap_year(year))
        printf("%d is a leap year.", year);
    else
        printf("%d is a nonleap year.", year);
    return 0;
}
```

1.3.1 Scope Rules in C

In Code 1.5, one may discover something surprising: the outermost `if` condition, i.e., `if(year % 100 == 0)` on line 5, is followed by a pair of brackets. However, the other `if` or `else` conditions contain a statement, but no brackets.

These can be explained through the scope rules in C. The C programming language has strict scope rules. Variables declared inside a function are not accessible outside the function. This is different from some other programming languages like Python or Java.

Both functions `is_leap_year` and `main` declare a variable with the identifier `year`. Variables declared inside a function or compound statement block are called *local variables*. A local variable can only be used in statements that are inside the function or block of code containing its declaration and *shadow* the definition of variables with the same identifier in an outer block of code. A local variable cannot be accessed outside of its scope. In addition, the variable `int year` in `is_leap_year` appears in the function definition, i.e., it is one of the function's *formal parameters*. In general, formal parameters are treated as local variables of the function.

1.3.2 (Optional) Recursion — Thinking Like Induction

Functions can be *recursive*, i.e., by calling itself in its declaration. A recursive function solves a problem by solving a smaller version of the same problem and combining the result. If you have seen mathematical induction, the parts line up:

- **Base case (induction's basis)**: The simplest input with a direct answer.
- **Recursive case (induction step)**: It assumes that the function works for a smaller input and use it to solve the current one.
- **Progress measure**: A number that strictly decreases toward the base case (e.g., n remaining length).

A good first example of recursion is the sum of the first n values of the Fibonacci sequence:

Code 1.6: `FibonacciRec.c`

```
#include <stdio.h>

int sum_prefix(const int *a, int n){
    if (n == 0) return 0; // Base case: sum of zero numbers is 0
```

```c
    return sum_prefix(a, n-1) + a[n-1]; // Recursive case: sum of first
        n is sum of first n-1 plus a[n-1]
}

int main(void){
    int a[] = {2, 4, 6};
    printf("%d\n", sum_prefix(a, 3)); // Prints "12"
    return 0;
}
```

Summarising from above, the recipe for writing correct recursion is to:

- Pick a **progress measure** that strictly decreases (e.g., `n`, `right-left`).
- Write a **base case** that returns immediately (no recursive call).
- In the **recursive case**, call yourself on the smaller input and combine the results.
- Ensure that every call moves closer to the base case (so it terminates).

Why should we avoid the naive Fibonacci here? The textbook recursive Fibonacci:

```c
int naiveFib(int n){
    if (n <= 1) return n;
    return naiveFib(n-1) + naiveFib(n-2);
}
```

makes exponentially many calls; it is slow and will eventually crash your program when `n` is too large, and is thus not a representative of good practice in C.

We will introduce another way, called **Loops**, to accomplish the same goal in Lab 2. Loops provide an efficient structure to calculate the Fibonacci sequence. Just to whet your appetite for Lab 2: an example loop-based Fibonacci is given in Code 1.7. Generally, loops are more efficient than recursions in most programming languages [6].

Code 1.7: `FibonacciLoop.c`

```c
#include <stdio.h>

int sum_prefix_iter(const int *a, int n) {
    int s = 0;
    for (int i = 0; i < n; ++i) s += a[i]; // A looping structure
    return s;
}
```

[6] Certain recursive functions, i.e., the tail-recursive functions, can be efficiently transformed to a loop structure. Tail-call optimization is out of scope for Introductory Programming and we mention it here only for those who are interested with some after-class exploration. Other considerations such as problem suitability and readability may also matter when it comes to choosing recursion vs. looping

```c

int main(void) {
    int a[] = {2, 4, 6};
    int n = (int)(sizeof a / sizeof a[0]);
    printf("%d\n", sum_prefix_iter(a, n)); // Prints 12
    return 0;
}
```

Takeaway: Recursion is a way to structure solutions with basic conditional statements and function calls. Use recursion when the problem naturally splits into smaller subproblems (e.g., a tree structure), and always specify the base case & progress measure. Looping is an alternative to recursion that can generally implement the same logic more efficiently in C.

1.4 Exercises A and B

Notation

The *time limit per test* below is the program execution time limit and your code will fail to pass the test case if the limit is exceeded. Please try your best to write efficient code.

A - Grade Point Average

Time limit per test: 1 second

Problem Statement

The grade point average (GPA) is a standard student performance metric used in Glasgow College and is calculated as a weighted average. To calculate a student's GPA you must:

1. Input how many courses a student has taken.
2. Input the credit and student grade for each course.
3. Use the weighted average to calculate and output the GPA.

The formula for calculating GPA is

$$\text{GPA} = \frac{a_1x_1 + a_1x_2 + \cdots + a_Nx_N}{a_1 + a_2 + \cdots + a_N}$$

where a_i and x_i indicate the credit and grade for the ith course, respectively.

Your task: given the grades of Calculus I, Introductory Programming, and an "anonymous" course. Calculate the GPA according to the formula above.

Input

The input has two lines.

The first line consists of 3 integers, which are the credits of Calculus I, Introductory Programming, and an "anonymous" course, respectively.

The second line also consists of 3 integers, which are the student's grades from Calculus I, Introductory Programming, and an "anonymous" course, respectively.

Output

Outputs an integer indicating the calculated GPA.

The final answer should be an integer **(rounded to the nearest integer)**.

Constraints

The grade points in the University of Glasgow range from 0 to 22.

Sample Input 1

```
2 1 3
22 18 13
```

Sample Output 1

```
17
```

Sample Input 2

```
10 5 3
20 12 9
```

Sample Output 2

```
16
```

Hint

You can use `round()` to round a floating point number to its nearest to. integer. To use this function you need to include the header file `math.h` in your C program.

Do you have any other mathematical method to do the same thing?

Base Code

```c
#include <stdio.h>

int main() {
    //TODO
}
```

B - Electrical Circuit

Time limit per test: 1 second

Problem Statement

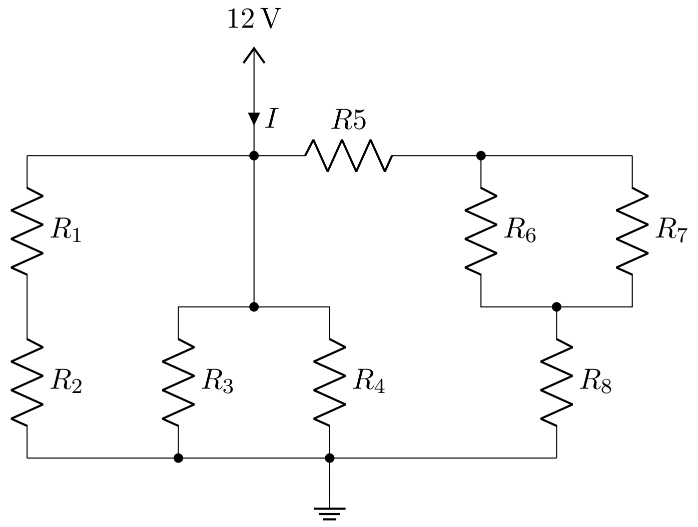
As electrical engineering students, you are familiar with Ohm's Law $U = IR$, in which U is the voltage across the resistor R and I is the current, that the equivalent resistance of two resistors in series is given by

$$R_s = R_1 + R_2,$$

and that the equivalent resistance of two resistors in parallel is given by

$$R_p = \left(\frac{1}{R_1} + \frac{1}{R_2}\right)^{-1}.$$

Consider the following circuit:



Given the voltage $U = 12\,\text{V}$ and a set of resistances $\{R_1, \ldots, R_8\}$, write a program to calculate the total current in this circuit. (The unit should be mA and output floating point numbers **two decimal** places.)

Input

The input consists of eight resistances: `R1 R2 R3 R4 R5 R6 R7 R8`

Output

Output the current I drawn from the power supply.

Constraints

The integer resistances have values between $1\,\Omega$ and $2\,\text{k}\Omega$, i.e.,

$$1 \leq R_i \in \mathbb{Z} \leq 2000,\ i \in \{1, 2, \ldots, 8\}.$$

Sample Input 1

```
100 200 300 400 500 600 700 800
```

Sample Output 1

```
117.39
```

Sample Input 2

```
123 234 345 456 567 678 789 890
```

Sample Output 2

```
101.30
```

Base Code

```c
#include <stdio.h>

// Please complete the function below and use these three functions to
    calculate the current.
// You may need to modify the function declarations to include all
    necessary formal parameters.

float series() {
    //TODO
}

float parallel() {
    //TODO
}

float Ohm_law() {
    //TODO
}

int main() {
    //TODO
}
```