

Computer Systems Architecture

Laboratory - Introduction to C

Radu-Tudor Vrînceanu
Bogdan Macovei

October 2026

Contents

1	Introduction	2
1.1	Introduction	2
2	Introduction to C	3
2.1	Introduction to the C Programming Language	3
2.2	<i>Hello, world!</i> in C	4
3	Fundamental Concepts of the C Language	6
3.1	Data Types and Pointers	6
3.1.1	Data Types	6
3.1.2	Pointer	7
3.2	Basic I/O Functions	8
3.2.1	<code>scanf</code>	8
3.2.2	<code>printf</code>	9
3.2.3	<code>fscanf</code>	9
3.2.4	<code>fprintf</code>	10
3.3	System Calls	12
3.3.1	<i>open</i> , <i>close</i> , and File Descriptors	12
3.3.2	What Does <code>return 0</code> Actually Mean?	13
3.4	Character Strings	14
3.5	Bitwise Operators	16
3.6	Passing Parameters by Reference and by Value	17
4	Proposed Exercises	19

1 Introduction

1.1 Introduction

The purpose of this material is to introduce you to the C programming language not from the perspective of syntax, but from that of the theoretical concepts that we will use later in future laboratory sessions. The motivation for choosing a language such as C lies in the ease with which we can manage internal memory and, at the same time, visualize its correspondence with GNU Assembly x86.

Note: If you are not familiar with certain elements of C syntax or have not studied the language before, please consult the materials from **Harvard CS50's Introduction to Computer Science**¹, which include both video materials and presentations covering the basic syntax of the language.

Once you know the basic syntax of C, after going through this material you will be able to better understand several language concepts that will be useful later:

- the *entry point* of a program, the `main(...)` function;
- basic I/O functions (*input / output*, `scanf`, `printf`, `fprintf`, `fscanf`);
- pointers;
- system calls;
- the *string.h* library for character strings;
- pass by reference and pass by value (*pass by reference*, *pass by value*);
- return by reference and return by value (*return by reference*, *return by value*).

For installing the C compiler, **GCC (GNU Compiler Collection)**, if you use a UNIX-based operating system (i.e. macOS, Linux-based), you will usually already have it installed. For those using Windows, if you have WSL installed, you can work within WSL. If you do not have WSL installed, we recommend following these steps for installation:

- download and run the installer for **MinGW (Minimalist GNU for Windows)**²;
- after the installation is complete, a new **MinGW Installation Manager** window will appear, and you must mark the *mingw32-gcc-g++* option for installation;
- save the selection and install the selected option by choosing **Installation > Apply Changes** from the upper-left corner, then confirm when the installation window appears by pressing **Apply**;
- in order to run the `gcc` command in **Command Prompt**, we must tell the operating system where it is located by editing the environment variables;
- on Windows, the environment variables window can be opened as follows: **Windows Explorer > right-click This PC > Advanced system settings > Environment Variables...**;

¹Harvard CS50's Introduction to Computer Science, <https://cs50.harvard.edu/x/2024/weeks/1/>

²MinGW (Minimalist GNU for Windows), <https://sourceforge.net/projects/mingw/>

- once in the environment variables window, look in the **System variables** table, usually the lower table, for a variable named **Path**;
- after identifying it, select it and click **Edit...** (be very careful here not to delete / overwrite the **Path** variable, as some programs may stop working afterward);
- a window named **Edit environment variable** will open; select the **New** option;
- in the newly created row, enter the path to the **bin** directory where **MinGW** was installed; if you kept the default installation path, enter **C:\MinGW\bin**;
- click **OK**; after closing the **Path** editing window, click **OK** again to close the environment variables window, and then one more **OK** for the advanced settings interface;
- open a **Command Prompt** and enter `gcc --version`; if you receive text displaying the compiler version, the installation is complete.

In this laboratory session, we will become more familiar with the foundations of the C programming language, as well as with the fundamental concepts that connect this language to GNU Assembly x86.

2 Introduction to C

2.1 Introduction to the C Programming Language

C is one of the oldest and most influential programming languages, developed by Dennis Ritchie at AT&T Bell Labs in 1972. Its original purpose was to provide a programming language that was efficient and flexible enough to be used in the development of the UNIX operating system, which was also created at Bell Labs.

Important characteristics of the C programming language:

- **Compiled language:** Code written in C is transformed into an executable file by a compiler, enabling fast program execution and code-level optimizations.
- **Static typing:** C is a statically typed language, which means that data types must be specified explicitly and are checked at compile time.
- **Procedural language:** C is based on the procedural and modular programming paradigm, allowing programmers to divide complex programs into smaller and more manageable functions.
- **Manual memory management:** Unlike modern languages that provide **garbage collectors**³, C leaves memory management to the programmer. This involves using functions such as *malloc* and *free* to manually allocate and release memory.

In conclusion, C has influenced many other modern languages, such as C++, Java, C#, and Objective-C. It is considered a mid-level language, offering a balance between the control provided by assembly languages and the complexity offered by high-level languages.

³garbage collectors, <https://craftinginterpreters.com/garbage-collection.html>

2.2 *Hello, world!* in C

In this program, we will learn what role the *main* function plays in receiving information from the outside and passing it to our program. A basic C program looks as follows:

```
#include <stdio.h>

int main()
{
    printf("Hello, world!\n");
    return 0;
}
```

After compiling and running this code, we will observe that in the *terminal*, or *cmd*, we obtain the message *Hello, world!* followed by a new line, due to the use of the special character `\n`.

- **UNIX Terminal:**

```
user@PC:~$ gcc main.c -o main
user@PC:~$ ./main
Hello, world!

user@PC:~$
```

- **Windows Command Prompt:**

```
C:\Users\user> gcc main.c -o main
C:\Users\user> main.exe
Hello, world!

C:\Users\user>
```

The *main* function represents the starting point on which the C *runtime* relies to execute a program. Normally, this function should have a series of parameters, and its complete form is the following:

```
int main(int argc, char** argv, char** arge)
```

- *argc* represents the number of arguments (i.e. *argument counter*);
- *argv* is an *array* of character strings representing the values of the *argc* arguments. *argv[argc]* = *NULL*, while *argv[0]* is the name of the executable program (i.e. *argument values*);
- *arge* is an *array* of character strings representing environment variables and their values (i.e. *argument environment*).

For a clearer view of these arguments, we will write the following code sequence in the *main.c* file:

```

#include <stdio.h>

int main(int argc, char** argv, char** arge)
{
    printf("%d\n", argc);

    for (int i = 0; i < argc; i++)
    {
        puts(argv[i]);
    }

    int e = 0;
    while (arge[e] != NULL)
    {
        puts(arge[e++]);
    }
    return 0;
}

```

We will explain the *printf* function in the next chapter. The *puts* function writes a character string to *stdout* (i.e. *standard output*), followed by `\n`. Thus, we can display both the argument values and the environment variables.

- **UNIX Terminal:**

```

user@PC:~$ gcc main.c -o main
user@PC:~$ arge0="Demo ENV Variable" ./main argv1 argv2 argv3
4
./main
argv1
argv2
argv3
... #other system ENV variables
arge0=Demo ENV Variable
... #other system ENV variables

```

- **Windows Command Prompt:**

```

C:\Users\user> gcc main.c -o main
C:\Users\user> set ENV="some value" & main.exe argv1 argv2 argv3 & set ENV=
4
./main
argv1
argv2
argv3
... #other system ENV variables
ENV="some value"
... #other system ENV variables

```

Note: *argc* is not used to determine how many environment variables were passed to the program; it only counts how many arguments were stored in *argv*.

3 Fundamental Concepts of the C Language

In this chapter, we will cover the most important functions and *I/O* concepts (i.e. the input and output data flow) in the C standard library. We will learn about the data types available in this language, dynamic memory allocation, character-string manipulation, and the mechanisms for passing variables to functions.

3.1 Data Types and Pointers

3.1.1 Data Types

Data types in C are classified into several groups, such as:

- **Primitive data types:** these are the basic data types used to represent simple values, such as *int*, *char*, *float*, *double*, *void*;
- **Derived data types:** these data types are built using primitive types or are *built-in*: *array*, *pointer*, *function*;
- **User-defined data types:** these are defined by the user, i.e. the person writing the code. In C, these types are represented by *struct*, *enum*, and *union*.

We will analyze the sizes of these data types, especially the primitive ones, using Microsoft's documentation for **Storage of basic types**⁴ and **C and C++ Integer Limits**⁵.

Observation: using the documentation, you will notice that there are several variants for representing an integer in memory, such as *short int* or *long long int*.

Observation: we can specify that the *MSB* (i.e. *Most Significant Bit*) should not be treated as a sign bit, but instead used as part of the number representation, by using the *unsigned* keyword.

Observation: the difference between double and float is reflected in the precision of the number approximation, since a computer with finite memory cannot represent something infinite.

```
#include <stdio.h>

int main() {
    short int x = 1;
    long long int y = 1;

    printf("%d %d", sizeof x, sizeof y);
    return 0;
}
```

Using the *sizeof* operator, we can see the difference in memory usage (represented in *Bytes*) between a *short* and a *long long*.

⁴Storage of basic types, <https://learn.microsoft.com/en-us/cpp/c-language/storage-of-basic-types?view=msvc-170>

⁵C and C++ Integer Limits, <https://learn.microsoft.com/en-us/cpp/c-language/cpp-integer-limits?view=msvc-170>

```

#include <stdio.h>

int main() {
    float x = 0.1, y = 0.2;

    printf("%.16f", x + y);
    return 0;
}

```

Run this code and observe what issue appears when the result of the addition is displayed. What happens if we replace the *float* type with *double*? All these aspects are related to a problem called *Floating point imprecision*, which you will discuss in detail during the course in the following weeks.

3.1.2 Pointer

In C, every variable is declared statically, i.e. memory is preallocated for that variable at the beginning of the program execution and deallocated at the end of the variable's scope.

If we do not want this to happen, we can use pointers in order to allocate memory when we need it. *A pointer is an address to a memory region.* A pointer can be declared as follows: `<data-type> * <pointer-name>.`

When we declare a pointer, the compiler allocates only 4 or 8 *bytes* (depending on the architecture), which are needed to store the memory address value. To allocate memory for a pointer using *malloc*:

```

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

int main() {
    int *p;

    p = (int *)malloc(sizeof(int));
    *p = 3;

    printf("Memory address %x, memory value %d", p, *p);
    return 0;
}

```

After defining the pointer and once we no longer need it in our program, we must deallocate it using *free*; otherwise, a phenomenon known as *Memory Leakage* would occur.

```

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

int main() {
    int *p;

    p = (int *)malloc(sizeof(int));
    *p = 3;
}

```

```

printf("Memory address %x, memory value %d", p, *p);

free(p);
return 0;
}

```

3.2 Basic I/O Functions

These basic functions for handling input and output data streams from *stdin* (i.e. *standard input*) and *stdout* (i.e. *standard output*) are found in the *stdio.h* library, which we have already included using the statement `#include <stdio.h>` at the beginning of the code.

3.2.1 scanf

Data is read using the function `int scanf(const char *format, ...)`. This function accepts a format parameter that specifies the type of data to be populated. The table below lists the most commonly used format strings:

Format	Description
<code>%d</code>	integer in base 10 (decimal)
<code>%i</code>	integer in base 10, 8, or 16, depending on its notation
<code>%u</code>	unsigned integer in base 10 (decimal), with the MSB taken into account
<code>%o</code>	integer in base 8 (octal)
<code>%x</code>	integer in base 16 (hexadecimal)
<code>%c</code>	character
<code>%s</code>	character string up to the first whitespace character
<code>%f</code>	real number

We will write the following code in *main.c*, where we can analyze how the `scanf` function is called:

```

#include <stdio.h>

int main(int argc, char** argv, char** arge)
{
    int x, y, z = 0;
    char c, name[256];

    scanf("%d %i %x %c %s", &x, &y, &z, &c, name);

    return 0;
}

```

Note: the `scanf` function is responsible for populating data while the program is running. This should not be understood as meaning that input data is read by passing arguments through *argv*.

Observation: after specifying the input format, we pass the variables in which we want to store the retrieved information. However, we can see that the `&` character is placed before the variable

names; this is the address-of operator and indicates the memory address of the variable. At the same time, we did not use this operator for the *name* variable, because `char name[256];` can be rewritten as `char* name;`. We know that pointers already refer to a memory address, so `&name` would result in an error.

3.2.2 printf

The function `int printf(const char *format, ...)` is similar to *scanf*, except that after passing the format string we no longer need to pass memory addresses, because we are interested in displaying the values of these variables.

```
#include <stdio.h>

int main(int argc, char** argv, char** arge)
{
    int x, y, z = 0, a = 8;
    float pi = 3.14159;
    char c, name[256];

    scanf("%d %i %x %c %s", &x, &y, &z, &c, name);
    printf("Dec: %d, Int: %i, Hexa: %x, Char: %c, String: %s\n", x, y, z, c, name);
    printf("Special formats: %02d, %.2f", a, pi);
    return 0;
}

$: ./main
010 010 0xbeef c ASC
Decimal: 10, Integer: 8, Hexadecimal: beef, Character: c, String: ASC
Special formats: 08, 3.14
```

Observation: although we provided the same value for both *x* and *y*, the `%i` format used for *y*, upon seeing a leading *0*, interpreted the value as being written in base 8 (octal). Thus, $10_8 = 8_{10}$, where n_b denotes the representation of *n* in base *b*.

3.2.3 fscanf

Sometimes we may not want to read data only from *stdin* or display it only to *stdout*, so we can use a function such as *fscanf* to read information from text files. It has the following prototype: `int fscanf(FILE *stream, const char *format, ...)`. We can see that it receives an additional parameter, namely a pointer to a type called *FILE*⁶, which is a structure defined in the *stdio.h* library.

Observation: In order to read from a file, the file must already exist on disk; otherwise, the read operation will fail. We can simulate this behavior by changing the name of the *input.txt* file after creating it.

```
// ''main.c
#include <stdio.h>
```

⁶FILE, <https://github.com/openbsd/src/blob/master/include/stdio.h#L99>

```

int main() {
    const char *filename = "input.txt";
    FILE *file = fopen(filename, "r"); // open it in read mode

    int number;
    char word[50];

    if (fscanf(file, "%d %16s", &number, word) == 2) {
        printf("Number: %d, Word: %s\n", number, word);
    } else {
        printf("Failed to read data from file.\n");
    }

    fclose(file); // it is very important to close the file after we have finished writing / reading
    return 0;
}

```

```
// '''input.txt
```

```
32 ASCIsMyFavoriteSubject
```

```
$: ./main
```

```
Number: 32, Word: ASCIsMyFavorite
```

3.2.4 fprintf

The function `int fprintf(FILE *stream, const char *format, ...)` is very similar to `printf`, except that this time we have an additional parameter denoting the file to which we will write the results produced from the format string.

Observation: In order to write to a file, it does **NOT** have to already exist on disk. To observe the behavior of the following code, we recommend deleting the previously created `input.txt` text file.

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

```

int main() {
    const char *filename = "input.txt";
    FILE *file = fopen(filename, "r");

    // if the file does not exist on disk in write mode, then the pointer will be null
    if (file == NULL) {
        file = fopen(filename, "w"); // open it in write mode
        if (file == NULL) {
            perror("Error creating file");
            return 1;
        }
        fprintf(file, "123 HelloWorld\n");
        fclose(file);
        file = fopen(filename, "r");
        if (file == NULL) {
            perror("Error opening file");
            return 1;
        }
    }
}

```

```

    }
}

int number;
char word[50];

if (fscanf(file, "%d %16s", &number, word) == 2) {
    printf("Number: %d, Word: %s\n", number, word);
} else {
    printf("Failed to read data from file.\n");
}

fclose(file);
return 0;
}

$: ./main
Number: 123, Word: HelloWorld

```

3.3 System Calls

In C, a *system call* is a special function provided by the *operating system kernel* (i.e. *OS kernel*) that allows a program to interact directly with hardware resources or to perform privileged actions that cannot be carried out solely from *user space*.

System calls in C are available through the *sys/syscall.h* library⁷. You can observe that each system call is assigned a code, i.e. a number, which will be useful later. The most common system calls are:

- *open*: opens a file for reading or writing;
- *read*: reads data from a file or device;
- *write*: writes data to a file or device;
- *close*: closes a file;
- *fork*: creates a new process copied from the process that made the call;
- *exec*: loads and executes another program in the address space of the current process;
- *wait*: waits for a child process to terminate;
- *exit*: terminates the execution of a program;
- *kill*: sends a signal to a process (for example, to terminate it).

3.3.1 *open*, *close*, and File Descriptors

The descriptor table. Each process has a descriptor table, consisting of an *array* of structures indexed from 0 to the largest possible descriptor. If the process has associated a descriptor *i* (a natural number) with a file, entry *i* in this table is filled with specific information, including a pointer to an entry in the open-file table. Usually, the first three descriptors are assigned automatically as follows: 0 = standard input, 1 = standard output, 2 = standard error output. They can also be designated by the following symbolic constants, defined in *unistd.h*: `STDIN_FILENO` (=0), `STDOUT_FILENO` (=1), `STDERR_FILENO` (=2).

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

int main()
{
    int file_descriptor;
    char buffer[100];

    // system call for opening a file
    file_descriptor = open("input.txt", O_RDONLY);
    if (file_descriptor < 0)
    {
```

⁷ *sys/syscall.h*, <https://github.com/openbsd/src/blob/master/sys/sys/syscall.h>

```

    perror("Error opening file");
    return 1;
}

// system call for reading data from the file
ssize_t bytes_read = read(file_descriptor, buffer, sizeof(buffer) - 1);
if (bytes_read < 0)
{
    perror("Read error");
    return 1;
}
buffer[bytes_read] = '\0'; // string termination

// display the content that was read
printf("Data read: %s\n", buffer);

// close the file
close(file_descriptor);
return 0;
}

```

3.3.2 What Does return 0 Actually Mean?

On a UNIX-like system, when *return 0* is executed from the `main()` function, the following occurs:

- control returns to the program entry point, *_start*, which is part of the C *runtime* for that system;
- the code in *_start* calls *exit(0)*, or something with the same effect;
- the *exit()* function invokes the *exit* system call, which transfers control to the kernel;
- the kernel reclaims all resources that the process was using;
- at some point, the parent process of the program calls *wait()* or a similar system call;
- the remaining records are cleaned up and the exit status (i.e. code) is reported.

3.4 Character Strings

Character strings are, in fact, character *arrays* that can be allocated dynamically or statically: `char *str = (char *)malloc(sizeof(char) * 10)` or, in the case of static allocation, `char str[10]`. Internally, *str* is actually a *pointer* to a *char*, because it indicates the memory address where this array is stored. Whenever we declare a character string, we must keep in mind that at its end we need to leave space available for the *null-terminated byte string* marker (i.e. `'\0'`). In C, there is a special library for manipulating these character strings called *string.h*.

In the following list, we will see several important functions from the *string.h* library that you will use when working with character strings.

- `size_t strlen(const char *s)`: obtains the length of a character string. **Note:** be careful, because *strlen* runs in linear time; use it outside iterative control structures whenever possible;
- `char *strcat(char *dest, const char *src)`: concatenates, i.e. joins, two character strings;
- `char *strcpy(char *dest, const char *src)`: copies one character string into another;
- `char *strchr(const char *s, int c)`: searches for the first occurrence of the character *c* in *s*;
- `char *strstr(const char *haystack, const char *needle)`: searches for the first occurrence of the substring *needle* in the character string *haystack*;
- `int strcmp(const char *s1, const char *s2)`: compares two character strings. If the strings are identical, the returned result is 0; otherwise, a negative value is returned if *s1* is lexicographically smaller than *s2*, and analogously a positive value may be returned;
- `char *strtok(char *str, const char *delim)`: extracts substrings from a character string, each delimited by characters from the *delim* string.

To read character strings from *stdin*, we can use the *gets* function; to read character strings from a file, we can use the *fgets* function. *scanf* is recommended only if the character string does not contain spaces. Output can be produced using either *puts* or *printf* to stdout; analogously, *fprintf* can be used for files.

Run this code to see how these functions from the *string.h* library work:

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

int main() {
    char *str = (char *)malloc(sizeof(char) * 128);
    puts("Enter a character string: ");

    gets(str);
    printf("The entered character string: %s\n", str);

    size_t length = strlen(str);
```

```

printf("Length of the character string: %d\n", length);

char *correct_dimension_str = (char *)malloc(sizeof(char) * (length + 1));

strcpy(correct_dimension_str, str);
printf("Copied character string: %s\n", correct_dimension_str);
printf("Are they identical?: %d\n", strcmp(str, correct_dimension_str));

strcat(str, " lorem, ipsum, dolor");
printf("Character string after concatenation: %s\n", str);
printf("Are they still identical?: %d\n", strcmp(str, correct_dimension_str));

for (size_t i = 0; i < length; i++) {
    if (strchr("aeiou", str[i])) {
        printf("We found the first vowel in the string at position %d, %c\n", i, str[i]);
        break;
    }
}

char *res = strstr(str, "ASC");
printf("Was the ASC string found? %s, sequence %s\n", (res != NULL ? "YES" : "NO"), res);

char sep[] = " ,";
char *token = strtok(str, sep);
while(token != NULL)
{
    puts(token);
    token = strtok(NULL, sep);
}

free(correct_dimension_str);
free(str);
return 0;
}

```

3.5 Bitwise Operators

In this section, we will present a table containing the most common bitwise operators and their meanings. You should know that any number can be represented in base 2 using a finite number of bits, from the perspective of the data types you use (i.e. declaring *short int* $x = 2$ is represented internally as *00000000 00000010*).

Name	Symbol
Bitwise OR operator	$x \mid y$
Bitwise AND operator	$x \& y$
Bitwise EXCLUSIVE OR operator	$x \wedge y$
Bitwise NOT operator	$\sim x$
Bitwise RIGHT and LEFT SHIFT operators	$x \ll y, x \gg y$

b_i^x	b_i^y	$(b_i^x \mid b_i^y)$	$(b_i^x \& b_i^y)$	$(b_i^x \wedge b_i^y)$
1	1	1	1	0
1	0	1	0	1
0	1	1	0	1
0	0	0	0	0

Here, b_i^x denotes the i -th bit in the binary representation of x , while b_i^y denotes the i -th bit in the binary representation of y .

Observation: $x \gg N$ means that we shift N bits to the right; if we consider what this means from the perspective of the binary representation, it is equivalent to $\lfloor x/2^N \rfloor$

Observation: $x \ll N$ means that we shift N bits to the left; if we consider what this means from the perspective of the binary representation, it is equivalent to $x \cdot 2^N$

Observation: $x \wedge x = 0$.

3.6 Passing Parameters by Reference and by Value

In a function, parameters can be passed in two ways:

- by value – before the call, a copy of the parameter is made and used throughout the execution of the function. Upon returning from the call, the parameter value remains unchanged: any modification made to the parameter inside the function body is not visible after the call is completed.
- by reference – the function is called directly with the variable passed as a parameter, and any modification made to that parameter inside the function body is visible after returning from the call. Constant values cannot be passed by reference to a function. A parameter is passed by reference if it has `&` before the parameter name.

However, there are certain types that are implicitly passed in a reference-like manner due to their underlying mechanism. In the example below, we can observe what happens to our string after running the code:

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

int by_value_and_reference(int x, double y, char *str) {
    x = 3;
    y = 3.141592653589793238;
    str[0] = 'B';

    printf("The value of str inside by_value is %s\n", str);
    printf("The value of y inside by_value is %.15f\n", y);
    return x;
}

int main(int argc, char *argv[], char *arge[]) {
    char *something_dynamic = (char*)malloc(sizeof(char*) * 4);
    strcpy(something_dynamic, "ASC");

    int x = 6; float pi = 3.1415;
    int result = by_value_and_reference(x, pi, something_dynamic);

    printf("The value of something_dynamic is %s\n", something_dynamic);
    printf("The value of pi is %.15f\n", pi);
    printf("The result is %d, and x is %d", result, x);

    free(something_dynamic);
    return 0;
}
```

Thus, we can observe that all primitive values are passed by value, while dynamically allocated variables are passed by reference based on the underlying mechanism. However, how can we pass a

primitive value by reference? We will rewrite the code above to allow the value of the variable *x* to be modified in the *by_value_and_reference* function.

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

int by_value_and_reference(int *x, double y, char *str) {
    *x = 3;
    y = 3.141592653589793238;
    str[0] = 'B';

    printf("The value of str inside by_value is %s\n", str);
    printf("The value of y inside by_value is %.15f\n", y);
    return *x;
}

int main(int argc, char *argv[], char *arge[]) {
    char *something_dynamic = (char*)malloc(sizeof(char*) * 4);
    strcpy(something_dynamic, "ASC");

    int x = 6; float pi = 3.1415;
    int result = by_value_and_reference(&x, pi, something_dynamic);

    printf("The value of something_dynamic is %s\n", something_dynamic);
    printf("The value of pi is %.15f\n", pi);
    printf("The result is %d, and x is %d", result, x);
    free(something_dynamic);
    return 0;
}
```

Observation: by passing *&x* as an argument, the type in the function header changes to *int **, because an address is being passed. We can therefore access its value through the dereference operator ***.

4 Proposed Exercises

- Write a C program that allows the values of two variables to be swapped without using an auxiliary variable.
- Write a C program that receives in a file a total of $2n + 1$ numbers, of which the $2n$ numbers are equal in pairs. Determine the number that has no pair by developing an algorithm that is optimal in terms of both space and execution time.
- Write a C program that receives through `argv` a string containing a date in the format `dd.mm.yyyy`; using the `sscanf` function, extract and display to `stdout` the day, month, and year (i.e. `./main Today, 02.10.2026, I attended the ASC laboratory session`)
- Write a C program that declares a *two-dimensional array* of size $N \times N$ (i.e. a square matrix) dynamically using `malloc` and `free`, filling the secondary diagonal with the numbers from $1 \dots N$.