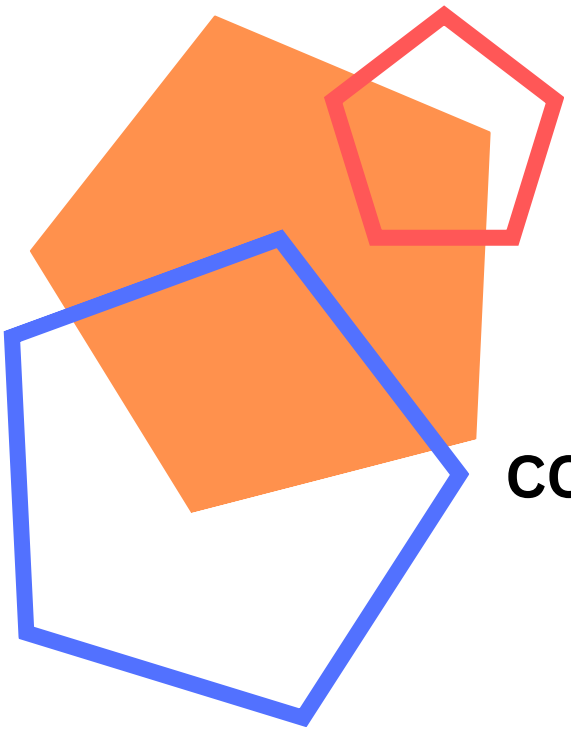
Teodora SANISLAV



COMPUTER PROGRAMMING AND PROGRAMMING LANGUAGES

Laboratory papers

RISOPRINT
Cluj-Napoca, 2024
ISBN 978-973-53-3271-6

Teodora SANISLAV

Computer Programming and Programming Languages

LABORATORY PAPERS

Editura RISOPRINT
Cluj-Napoca, 2024

*Editura **RISOPRINT** este recunoscută de **C.N.C.S.***
***(Consiliul Național al Cercetării Științifice)**.*
www.risoprint.ro ***www.cncs-uefiscdi.ro***

ISBN 978-973-53-3271-6

Computer Programming and Programming Languages

Autor
TEODORA SANISLAV

Director editură: GHEORGHE POP

Contents

List of Figures

List of Tables

Listings

Laboratory paper 1

C Input/Output (I/O)

1.1 Overview

- Presentation of the input (I) and output (O) built-in library functions
- Use of the I/O functions within simple C programs
- Work time: 2 hours

1.2 Theoretical Considerations

Input (I) refers to supplying data to a program, which can be done through a file or by entering a sequence of characters in the command line. **Output** (O) denotes presenting data on a screen, printer, or saving it in a file.

In the C programming language, all I and O operations are handled through streams (files), which consist of sequences of characters organized into lines. According to ANSI C standards each line must be at least 254 characters long and concludes with the '\n' ("new-line" character). When a C program runs, the following three streams are automatically opened to facilitate access to the keyboard (for reading data) and screen (for displaying data):

- Standard input stream (**stdin**) – connected to the keyboard;
- Standard output stream (**stdout**) – connected to the screen;
- Standard error stream (**stderr**) – connected to the screen.

The C programming language provides various built-in library functions for performing I/O tasks, which are listed in the <**stdio.h**> header file. The most commonly used are the following:

- For input: scanf, sscanf, getchar, gets;
- For output: printf, sprintf, putchar, puts.

1.2.1 scanf and printf

The **scanf** function reads the input data from the standard input stream (stdin), converts the data to their corresponding internal representations according to the format provided, and stores the obtained representations of data in the variables provided as arguments. The end of the input is indicated when the ENTER key is pressed [1].

The prototype of the **scanf** function is:

```
1 int scanf(const char *format [, &variable1] [, &variable2] ...);
```

scanf returns the number of items that were successfully read.

The format for **scanf** is specified as a character string enclosed in double quotes ("). The format string includes format specifiers, which define the conversion rules and have the following form: **%[*][width]type**, where:

- * (optional) – character that signifies that the input data read from stdin are not assigned to any variable;
- width (optional) – decimal number which defines the maximum length of the data to be read;
- type – one or two characters which define the data type of the conversion result.

The characters defining the type of conversion data are given in Table 1.1.

The & character before a variable name provides the address of that variable, thus telling the system where to store the input data. There is no & character before a variable name, if the variable is a single dimensional (one-dimensional) array (i.e., character string).

Table 1.1: Characters of format specifiers and their corresponding data types

Character	**Data type**
c	Signed character (char).
s	Character string (char *).
d	Integer as a signed decimal number (int).
u	Integer as an unsigned decimal number (unsigned int).
o	Integer as an octal number.
x, X	Integer as a hexadecimal number.
hd, hu	Integer as a short decimal number (short int), unsigned short decimal number (unsigned short int).
ld, lo, lx, lX	Integer as long decimal (long int), long octal, long hexadecimal.
lu	Integer as a long unsigned decimal number (unsigned long int).
f	Floating-point number (float).
lf	Double floating-point number (double).
Lf	Long double floating-point number (long double).

The **printf** function converts the data into their corresponding external representations based on the provided format and writes the output data to the standard output stream (stdout) [1].

The prototype of the **printf** function is:

```
int printf(const char *format [, expression list]);
```

printf returns the number of characters that were successfully written to the output.

The format for **printf** is defined as a character string enclosed in double quotes ("). This format string includes character sequences that will be displayed along with format specifiers. A format specifier follows the pattern: **%[flags][width][.precision]type**, where:

- flag (optional) – has the values:
  - '-' which specifies that the data to be displayed will be left-justified (the default justification is right-justification);
  - '+' which mandates that the displayed data be preceded by a + or - sign, even for positive numbers;
  - '#' which forces to precede the displayed data with 0, 0x or 0X when it is used with o, x or X type for values different than zero;
  - '0' which left-pads the displayed data with zeros instead of spaces;
- width (optional) – decimal number which indicates the minimum length of the field which will hold the displayed data;
- precision (optional) – decimal number specifying the precision;
- type – one or two characters which define the data type of the conversion result. In addition to the characters used by the **scanf** function, characters 'e' or 'E' can also be used for displaying floating point data (in single or double precision) in scientific notation, as well as 'p' for displaying the pointer data type in hexadecimal.

Listing 1.1 presents several calls of the **scanf** and **printf** functions.

```
#include <stdio.h>
int main() {
    char  test_char       = '\0';
    char  test_string[10] = "";
    int   test_integer    = 0;
```

```
    float test_float      = 0.0f;

    printf("Enter a character: ");
    scanf("%c", &test_char);
    printf("Character entered = %c\n", test_char);

    printf("Enter an array of characters: ");
    scanf("%s", test_string);
    printf("Array of charaters entered = %s\n", test_string);

    printf("Enter an integer: ");
    scanf("%d", &test_integer);
    printf("Integer value entered = %d\n", test_integer);
    printf("Octal value = %#o, Hexadecimal value = %#X\n", test_integer,
    test_integer);

    printf("Enter a float number: ");
    scanf("%f", &test_float);
    printf("Float value entered = %f\n", test_float);
    return 0;
}
```

Listing 1.1: Program L1Ex1.c

Listing 1.2 presents several calls of the **printf** function which highlight different formats for displaying integer and float values.

```
#include <stdio.h>
int main() {
    int   integer_no = 45678;
    float float_no   = 456.789f;

    printf("45678 right justified to 6 columns: %6d\n", integer_no);
    printf("456.789 rounded to 2 digits: %.2f\n", float_no);
    printf("456.789 rounded to 0 digits: %.f\n", float_no);
    printf("456.789 in exponential form: %e\n", float_no);
    printf("456.789 right justified to 8 columns and rounded to 2 digits:
    %8.2f\n", float_no);
    return 0;
}
```

Listing 1.2: Program L1Ex2.c

1.2.2 sscanf and sprintf

The **sscanf** function reads formatted input from a character string [2]. Unlike the **scanf** function, **sscanf** has an additional first argument that indicates the character string from which the data are read, instead of using the standard input stream (stdin).

The prototype of the **sscanf** function is:

```
int sscanf(const char *str, const char *format [, &variable1] [, &variable2]
    ...);
```

sscanf returns the number of variables that were successfully filled.

The **sprintf** function sends formatted output to a character string [2]. Unlike the **printf** function, **sprintf** includes an additional first argument that indicates the character string where the data will be written, rather than directing the output to the standard output stream (stdout).

The prototype of the **sprintf** function is:

```
int sprintf(char *str, const char *format [, expression list]);
```

sprintf returns the total number of characters written, excluding the '\0' (NULL character) that is appended at the end of the character string [2].

Listing 1.3 presents several calls of the **sscanf** and **sprintf** functions.

```
#include <stdio.h>
int main() {
    char  *input = "2 4 5.25", output[100] = "";
    int   a = 0, b = 0;
    float c = 0.0f, sum = 0.0f;

    sscanf(input, "%d%d%f", &a, &b, &c);
    printf("a = %d b = %d c = %.2f\n", a, b, c);
    sum = a + b + c;
    sprintf(output, "%.2f", sum);
    printf("Sum of a, b and c is %s.\n", output);
    return 0;
}
```

Listing 1.3: Program L1Ex3.c

1.2.3 getchar and putchar

The **getchar** function reads a character from the standard input stream (stdin) and returns it as an integer (the ASCII code of the character) [1]. The end of the input is indicated when the ENTER key is pressed.

The prototype of the **getchar** function is:

```
int getchar(void);
```

getchar returns the character read as an unsigned character cast to an integer upon success.

The **putchar** function writes the character given as an argument (the ASCII code of the character) on the standard output stream (stdout) [1].

The prototype of the **putchar** function is:

```
int putchar(int char);
```

putchar returns the character written as an unsigned character cast to an integer on success.

Listing 1.4 exemplifies the use of the **getchar** and **putchar** functions.

```
#include <stdio.h>
int main () {
    int ch = 0;

    printf("Enter a character: ");
    ch=getchar();
    printf("You entered: ");
    putchar(ch);
    printf("\nASCII code of %c: %d\n", ch, ch);
    return 0;
}
```

Listing 1.4: Program L1Ex4.c

1.2.4 gets and puts

The **gets** function reads a line from the standard input stream (stdin) and stores it into a character string given as its argument [1]. It stops reading when either the '\n' ("newline" character) is encountered or the end-of-file is reached.

The prototype of the **gets** function is:

```
char *gets(char *str);
```

gets returns a character string on success.

The **puts** function writes a character string given as its argument to the standard output stream (stdout) [1].

The prototype of the **puts** function is:

```
int puts(const char *str);
```

puts returns a non-negative value upon succes.

Listing 1.5 exemplifies the use of the **gets** and **puts** functions.

```
#include <stdio.h>
int main() {
    char name[50] = "";
    char cp[100]  = "";

    printf("What is your name?\n");
    gets(name);
    printf("------------------------------\n");
    printf("Well, %s, what do you study?\n", name);
    gets(cp);
    printf("------------------------------\n");
    puts("Let me see if I got it:");
    puts(name);
    puts("studies");
    puts(cp);
    return 0;
}
```

Listing 1.5: Program L1Ex5.c

1.3 Lab Tasks

1. Run the programs given as examples and analyze the output.
2. Identify and fix the errors in each of the statements below:
   a) int int_num = 0; printf("The value is %d\n", &int_num);
   b) int int_num1 = 0, int_num2 = 0; scanf("%d%d", &int_num1, int_num2);
   c) int x = 0, y = 0; printf("The sum of %d and %d is %d.\n, x, y);
   d) double x = 0.00, y = 0.00; print("The product is %d.\n," x * y);
   e) gets();
   f) int int_num1 = 0, int_num2 = 0; sscanf("%d%d", &int_num1, &int_num2);
   g) getchar(10);
3. Write a C program to output the Euler's number $e = 2.718281$ using various floating point format specifiers (4 calls of the printf function are required).
4. Write a C program to present the integer values from 0 to 15 in three columns, showing their decimal, octal, and hexadecimal representations.
5. Write a C program to display the characters that are on the keyboard.

6. Write a C program to display the following:

```
    1 12345
   12 1234
  123 123
 1234 12
12345 1
```

7. Write a C program to calculate and display the product of three integers read from the keyboard.

8. Write a C program to calculate and display the result of addition, subtraction, multiplication, division and average of two real numbers read from the keyboard.

1.4 References

1. Iosif Ignat, *Programarea calculatoarelor: indrumator de lucrari de laborator*, 2003, Second edition, U.T.Press, Cluj-Napoca, ISBN: 973-662-024-7.
2. Paul Deitel, Harvey Deitel, *C How to Program*, 2022, Ninth edition, Pearson Education, ISBN: 978-0-13-739839-3.

Laboratory paper 2

Data Types and Expressions in C

2.1 Overview

- Presentation of basic data types in C
- Presentation of expressions in C
- Presentation of C operators
- Presentation of data types default conversions
- Use of expressions and operators within simple C programs
- Work time: 2 hours

2.2 Theoretical Considerations

2.2.1 Basic data types in C

A data type is a format for data storage that encompasses a range of values. It is used to define variables, constants, arrays, pointers or functions before they can be utilized in a program. Table 2.1 presents the C programming language basic data types (char, int, float, double), as defined in the C89/C99/C11/C17/C23 versions of the C standard. These standards do not define the size for each data type, but only the range, and the space allocated in the computer's memory depends on the system.

Table 2.1: C basic data types

	32-bit machine	
Data type	**Size**	**Range**
char signed char	1 byte (8 bits)	-128 to 127 (-2^{8-1} *to* $2^{8-1}-1$)
unsigned char	1 byte (8 bits)	0 to 255 (0 *to* $2^{8}-1$)
short short int signed short signed short int	2 bytes (16 bits)	-32768 to 32767 (-2^{16-1} *to* $2^{16-1}-1$)
unsigned short unsigned short int	2 bytes (16 bits)	0 to 65535 (0 *to* $2^{16}-1$)
int signed signed int	4 bytes (32 bits)	-2147483648 to 2147483647 (-2^{32-1} *to* $2^{32-1}-1$)
unsigned unsigned int	4 bytes (32 bits)	0 to 4294967295 (0 *to* $2^{32}-1$)
long long int signed long signed long int	4 bytes (32 bits)	-2147483648 to 2147483647 (-2^{32-1} *to* $2^{32-1}-1$)
unsigned long unsigned long int	4 bytes (32 bits)	0 to 4294967295 (0 *to* $2^{32}-1$)
float (single-precision)	4 bytes (32 bits)	1.175E-038 to 3.403E+038 (6 digits precision)
double (double-precision)	8 bytes (64 bits)	2.225E-308 to 1.798E+308 (15 digits precision)
long double (extended-precision)	10 bytes (80 bits) or 12 bytes (96 bits) or 16 bytes (128 bits)	

The <**limits.h**> header file includes definitions of the properties (i.e., range) of the char and int data types. Some of them are [1]:

- CHAR_BIT – size of the char type in bits;
- CHAR_MIN, CHAR_MAX– minimum and maximum possible values for the char type;

- MB_LEN_MAX – maximum number of bytes in a multibyte character;
- SCHAR_MIN, SHRT_MIN, INT_MIN, LONG_MIN – minimum possible values for the signed char and signed integer types;
- SCHAR_MAX, SHRT_MAX, INT_MAX, LONG_MAX – maximum possible values for the signed char and signed integer types;
- UCHAR_MAX, USHRT_MAX, UINT_MAX, ULONG_MAX – maximum possible values for the unsigned char and unsigned integer types.

The <**float.h**> header file includes definitions of the properties (i.e., range, precision) of floating-point data types. Some of them are (note that in all cases, FLT refers to float, DBL refers to double, and LDBL refers to long double) [1]:

- FLT_MIN, DBL_MIN, LDBL_MIN – minimum normalized positive values for the floating-point types;
- FLT_MAX, DBL_MAX, LDBL_MAX – maximum finite values for the floating-point types;
- FLT_DIG, DBL_DIG, LDBL_DIG – number of decimal digits that can be represented without losing precision by the floating-point types;
- FLT_RADIX – radix of the exponent in the floating-point types;
- FLT_MANT_DIG, DBL_MANT_DIG, LDBL_MANT_DIG – number of digits in the floating-point significand.

Listing 2.1 computes the values presented in Table 2.1.

```c
#include <stdio.h>
#include <limits.h>
#include <float.h>
int main() {
  printf("-------------------------------------------------------------------------\n");
  printf("|Data type\t\t| Bytes\t| Range\t\t\t\t|\n");
  printf("-------------------------------------------------------------------------\n");

  printf("|char\t\t\t| %u\t| %d to %d\t\t\t|\n", sizeof(char),SCHAR_MIN,SCHAR_MAX);
  printf("|unsigned char\t\t| %u\t| %d to %d\t\t\t|\n", sizeof(unsigned char), 0,
    UCHAR_MAX);
  printf("-------------------------------------------------------------------------\n");

  printf("|short int\t\t| %u\t| %d to %d\t\t|\n", sizeof(short int), SHRT_MIN,
    SHRT_MAX);
  printf("|unsigned short int\t| %u\t| %d to %d\t\t\t|\n", sizeof(unsigned short
    int), 0, USHRT_MAX);
  printf("-------------------------------------------------------------------------\n");

  printf("|int\t\t\t| %u\t| %d to %d\t|\n", sizeof(int), INT_MIN, INT_MAX);
  printf("|unsigned int\t\t| %u\t| %d to %u\t\t|\n", sizeof(unsigned int), 0,
    UINT_MAX);
  printf("-------------------------------------------------------------------------\n");

  printf("|long int\t\t| %u\t| %d to %d\t|\n", sizeof(long int),LONG_MIN,LONG_MAX);
  printf("|unsigned long int\t| %u\t| %d to %u\t\t|\n", sizeof(unsigned long int),
    0, ULONG_MAX);
  printf("-------------------------------------------------------------------------\n");

  printf("|float\t\t\t| %u\t| %.3e to %.3e\t|\n", sizeof(float), FLT_MIN, FLT_MAX);
  printf("|double\t\t\t| %u\t| %.3e to %.3e\t|\n", sizeof(double),DBL_MIN,DBL_MAX);
  printf("|long double\t\t| %u\t| %.3e to %.3e\t|\n", sizeof(long double), LDBL_MIN
    , LDBL_MAX);
  printf("Precision of float: %u digits.\n", FLT_DIG);
  printf("Precision of double: %u digits.\n", DBL_DIG);
  printf("Precision of long double: %u digits.\n", LDBL_DIG);
  return 0;
}
```

Listing 2.1: Program L2Ex1.c

Appendix - Data Representation in Computer Memory shows how the data related to the basic data types of the C programming language are represented in the computer's memory.

2.2.2 Expressions in C

C **expressions** consist of one or more **operands** joined by **operators**.

An **operand** has a data type associated with it and a corresponding value. An operand can be a constant, the name of a variable (scalar or array), an element of an array, the name of a structure, a reference to an element of a structure, the name of a data type, the name of a function, a function call, or an expression surrounded by brackets.

An **operator** is a symbol which performs various operations (i.e., logical, mathematical) [2].

Several C expressions are provided in the following listing.

```
a = b + 3; // a, b and 3 are operands; = and + are operators
++z; // z is an operand; ++ is an operator
300 > (8 * k[5]); // 300, 8, k[5] and (8 * k[5]) are operands; > and * are
    operators
```

An expression is evaluated according to:

- Operators priority and associativity;
- Operands data types default conversions.

2.2.3 Operators in C

C offers many types of operators that can be classified as follows: arithmetic, increment/decrement, relational, logical, bitwise, assignment, conditional, and special operators (Table 2.2). Some of them are unary, other are binary, and one of them is ternary.

Table 2.2: C operators

Type of operator	Description
Arithmetic	Used to perform mathematical calculations (+,-,*,/,%).
Increment/decrement	Used to increase or decrease the value of the variable by one (++,−−).
Relational	Used to compare the value of two given variables (<,>,<=,>=,==,!=).
Logical	Used to perform logical operations on two given variables (&&,||,!).
Bitwise	Used to perform bit operations on given two variables (&,|,∼,^,«,»).
Assignment	Used to assign values for the variables (=,+=,-=,*=,/=,%=,&=,^=, |=, «=, »=).
Conditional	Returns a value if condition is true and returns another value if condition is false. (Condition ? true_value: false_value)
Special	(), [], &, *, ., „->, sizeof( ), (type-name)

Arithmetic operators

Table 2.3 presents the arithmetic operators available in C and their exemplification using variable a (first operand) with a value of 10 and variable b with a value of 20 (second operand).

Table 2.3: C arithmetic operators [2]

Operator	Description	Example
+	Addition operator Adds two operands.	a + b = 30
-	Substraction operator Subtracts the second operand from the first operand.	a - b = -10
*	Multiplication operator Multiplies the first operand with the second operand.	a * b = 200
/	Division operator Divides the first operand by the second operand.	b / a = 2
%	Modulus operator Computes the remainder after dividing the first operand by the second operand.	a % b = 10

Increment/decrement operators

The increment operator is used to increase the value of an operand by one and the decrement operator is used to decrease the value of an operand by one. They can be prefix (++operand) or postfix (operand−−) (Table 2.4).

Table 2.4: C increment/decrement operators [2]

Operator	Description	Example
++	Increment operator Increments the value of an operand by 1. Prefix - the value after increment is used. Postfix - the value before increment is used.	 ++a a++
−−	Decrement operator Decrements the value of an operand by 1. Prefix - the value after decrement is used. Postfix - the value before decrement is used.	 −−a a−−

Listing 2.2 presents the use of the prefix and postfix increment and decrement operators.

```c
#include<stdio.h>
int main() {
  int a = 2, b = 0;

  b = a++ + a-- + ++a + --a;
  printf("Value of a is %d and value of b is %d.", a, b);
  return 0;
}
```

Listing 2.2: Program L2Ex2.c

Relational operators

Table 2.5 presents the relational operators available in C and their exemplification using variable a (first operand) with a value of 10 and variable b with a value of 20 (second operand). C evaluates all expressions that contain relational operators to 0 (false) or 1 (true).

Table 2.5: C relational operators [2]

Operator	Description	Example
==	Equal to operator Verifies if the values of two operands are equal. If yes, then the condition becomes true.	(a == b) is false
!=	Not equal operator Verifies if the values of two operands are not equal. If yes, then the condition becomes true.	(a != b) is true
>	Greater than operator Verifies whether the value of the left operand is greater than the value of the right operand. If yes, then the condition becomes true.	(a >b) is false
<	Less than operator Verifies whether the value of the left operand is less than the value of the right operand. If yes, then the condition becomes true.	(a <b) is true
>=	Greater than or equal to operator Verifies whether the value of the left operand is greater than or equal to the value of the right operand. If yes, then the condition becomes true.	(a >= b) is false
<=	Less than or equal to operator Verifies whether the value of the left operand is less than or equal to the value of the right operand. If yes, then the condition becomes true.	(a <= b) is true

Logical operators

Table 2.6 presents the logical operators available in C and their exemplification using variable a (first operand) with a value of 10 and variable b with a value of 0 (second operand). C evaluates all expressions that contain logical operators to 0 (false) or 1 (true).

Table 2.6: C logical operators [2]

Operator	Description	Example
&&	Logical AND operator If both operands are non-zero, the condition evaluates to true.	(a && b) is false
||	Logical OR operator If either of the two operands is non-zero, the condition evaluates to true.	(a || b) is true
!	Logical NOT operator It is used to reverse the logical state of its operand.	!(a && b) is true

Bitwise operators

Table 2.7 presents the bitwise operators available in C. These operators work on bits and execute bit-by-bit operations. Their exemplification using variable a (first operand) with a value of 10 and variable b with a value of 0 (second operand) is also presented in Table 2.7.

Table 2.7: C bitwise operators [3]

Operator	Description	Example
&	Bitwise AND operator The bits in the result are set to 1 if the corresponding bits in both operands are both 1.	(a & b) = 0
|	Bitwise OR operator The bits in the result are set to 1 if at least one of the corresponding bits in either of the two operands is 1.	(a | b) = 10
^	Bitwise exclusive OR operator The bits in the result are set to 1 if exactly one of the corresponding bits in the two operands is 1.	(a ^ b) = 10
~	Bitwise (1's) complement operator All 0 bits are set to 1 and all 1 bits are set to 0.	(~a) = -11
«	Left shift Shifts the bits of the first operand left by the number of bits specified by the second operand. One left shift multiplies by 2 the first operand.	(a « 2) = 40
»	Right shift Shifts the bits of the first operand right by the number of bits specified by the second operand. One right shift divides by 2 the first operand.	(a » 2) = 2

Listing 2.3 presents the use of the bitwise AND operator to verify if an integer is even or odd.

```c
#include <stdio.h>
int main() {
  unsigned int integer_no = 0;

  printf("Enter an unsigned integer: ");
  scanf("%u", &integer_no);
  if(integer_no & 1)
    printf("%u is odd.", integer_no);
  else
    printf("%u is even.", integer_no);
  return 0;
}
```

Listing 2.3: Program L2Ex3.c

Assignment operators

Table 2.8 presents the assignment operators available in C and their exemplification using the variables *a* and *b*.

Table 2.8: C assignment operators [2]

Operator	Description	Example
=	Simple assignment operator Assigns values from right side operand to left side operand.	a = b
+=	Add and assignment operator Adds the right operand to the left operand and assigns the result to the left operand.	a += b is equivalent to a = a + b
-=	Subtract and assignment operator Subtracts the right operand from the left operand and assigns the result to the left operand.	a -= b is equivalent to a = a - b
*=	Multiply and assignment operator Multiplies the right operand with the left operand and assigns the result to the left operand.	a *= b is equivalent to a = a * b
/=	Divide and assignment operator Divides the left operand with the right operand and assigns the result to the left operand.	a /= b is equivalent to a = a / b
%=	Modulus and assignment operator Takes modulus using two operands and assigns the result to the left operand.	a %= b is equivalent to a = a % b
«=	Left shift and assignment operator	a «= b is equivalent to a = a « b
»=	Right shift and assignment operator	a »= b is equivalent to a = a » b
&=	Bitwise AND and assignment operator	a &= b is equivalent to a = a & b
|=	Bitwise OR and assignment operator	a |= b is equivalent to a = a | b
^=	Bitwise exclusive OR and assignment operator	a ^= b is equivalent to a = a ^ b

Conditional operator

The conditional operator (Table 2.9) is the only ternary operator in C. It can be used as a shortcut for an if ... else statement. It is typically used as part of a larger expression where an if ... else statement would be awkward.

Table 2.9: C conditional operator

Operator	Description	Example
? :	Conditional operator If condition is true ? then value X : otherwise value Y.	b = (a == 10) ? 20 : 30

Listing 2.4 presents the use of the conditional operator to verify whether an integer is even or odd.

```c
#include <stdio.h>
int main() {
  int integer_no = 0;

  printf("Enter an integer: ");
  scanf("%d", &integer_no);
  (integer_no % 2 == 0) ? printf("Even integer.") : printf("Odd integer.");
  return 0;
}
```

Listing 2.4: Program L2Ex4.c

Special operators

Table 2.10 shows the special operators available in C.

Table 2.10: C special operators

Operator	Description	Example
sizeof()	Returns the size of a variable in bytes.	sizeof(a), where a is an integer, returns 4
&	Reference operator Returns the address of a variable.	&a
*	Dereference operator Returns the value stored at a particular address.	*a
(type-name)	Cast operator Performs typecasting.	(float)a, where a is an integer
()	Functional call operator	
[]	Array element reference (subscripting) operator	
.	Direct member selection operator	
–>	Indirect member selection operator	
,	Separator of expressions	

Operators precedence and associativity

There are rules that govern how the expressions in C get interpreted by the compiler. Operators precedence and associativity rules are some of them. The precedence of the operators determines their rank. The associativity determines the order of performing the operations in which there are operators with the same precedence. Table 2.11 shows, from highest to lowest, the C operators' order of precedence and their associativity.

2.2.4 Data types default conversions

When expressions that contain operands of different data types are evaluated, the data types of the operands are converted. These conversions can be performed implicitly or explicitly:

- The implicit conversion, also called "type promotion", is done automatically by the compiler without the intervention of programmers. The compiler converts the operands of the expressions into the data type of the largest operand;
- The explicit type conversion is performed by the programmers who can tell the compiler to treat a value as a certain data type, using the cast operator. To prevent information loss during the conversion of expressions from one data type to another, the following rules must be taken into consideration: all characters must be converted to integers, all integers must be converted to floats, and all floats must be converted to doubles.

Table 2.11: C operators precedence and associativity [3, 4]

Precedence	**Operator**	**Meaning of operator**	**Associativity**
1	()	Functional call	Left to right
	[]	Array subscripting	
	->	Indirect member selection	
	.	Direct member selection	
	++, --	Postfix increment and decrement	
2	!	Logical NOT	Right to left
	∼	Bitwise(1's) complement	
	+	Unary plus	
	-	Unary minus	
	++, --	Prefix increment and decrement	
	&	Reference	
	*	Dereference	
	sizeof()	Returns the size of a variable	
	(type-name)	Cast	
3	*	Multiplication	Left to right
	/	Division	
	%	Modulus	
4	+	Addition	Left to right
	-	Subtraction	
5	«	Left shift	Left to right
	»	Right shift	
6	<	Less than	Left to right
	<=	Less than or equal to	
	>	Greater than	
	>=	Greater than or equal to	
7	==	Equal to	Left to right
	!=	Not equal	
8	&	Bitwise AND	Left to right
9	^	Bitwise exclusive OR	Left to right
10	|	Bitwise OR	Left to right
11	&&	Logical AND	Left to right
12	||	Logical OR	Left to right
13	?:	Conditional operator	Right to left
14	=	Simple assignment	Right to left
	*=	Multiply and assignment	
	/=	Divide and assignment	
	%=	Modulus and assignment	
	+=	Add and assignment	
	-=	Substract and assignment	
	&=	Bitwise AND and assignment	
	^=	Bitwise exclusive OR and assignment	
	|=	Bitwise OR and assignment	
	«=	Left shift and assignment	
	»=	Right shift and assignment	
15	,	Separator of expressions	Left to right

Listing 2.5 presents the explicit type conversion and emphasizes the difference between floating-point division and integer division.

```c
#include<stdio.h>
int main() {
  int   integer_no = 5;
  float float_no   = integer_no / 6;

  printf("float_no = %.2f\n", float_no);
  float_no = (float)integer_no / 6;
  printf("float_no = %.2f\n", float_no);
  return 0;
}
```

Listing 2.5: Program L2Ex5.c

2.3 Lab Tasks

1. Run the programs given as examples and analyze the output.
2. Give examples of unary, binary and ternary operators.
3. What will be the output of the program?

```c
#include <stdio.h>
int main() {
  int value, number = 50;

  value = (number > 30 ? (number < 60 ? 50 : 100): 150);
  printf("%d\n", value);
  return 0;
}
```

A. 150
B. 100
C. 30
D. 50

4. What will be the output of the program?

```c
#include <stdio.h>
int main() {
  int i = 3, j = 0, k = 0;

  j = 2 * (i++);
  k = 2 * (++i);
  printf ("i = %d j = %d k = %d", i, j, k);
  return 0;
}
```

A. i = 3 j = 8 k = 8
B. i = 5 j = 8 k = 10
C. i = 5 j = 6 k = 10
D. i = 4 j = 8 k = 8

5. What will be the output of the program?

```c
#include <stdio.h>
int main() {
  int x = 20, y = 35;

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

A. x = 58 y = 93
B. x = 57 y = 94
C. x = 56 y = 93
D. x = 57 y = 84

6. What will be the output of the program?

```c
#include <stdio.h>
int main() {
  unsigned int x = 5;

  printf("x is %u, x << 2 is %u, x >> 2 is %u", x, x << 2, x >> 2);
  return 0;
}
```

A. x is 5, x « 2 is 21, x » 2 is 1
B. x is 5, x « 2 is 20, x » 2 is 1
C. x is 5, x « 2 is 19, x » 2 is 0
D. x is 5, x « 2 is 19, x » 2 is 1

7. Assuming i = 1, j = 2, k = 3, and m = 4, what does each of the following expressions return?
a) printf("%d", m == 1);
b) printf("%d", i > 2 && j < 4);
c) printf("%d", m < 10 && k < m);
d) printf("%d", m >= j || k == i);
e) printf("%d", (i + j >= m) || (3 - k > j));
f) printf("%d", !k);
g) printf("%d", !(j - i));
h) printf("%d", !(k > m));

8. Write a C program to convert a real number, representing a measurement for an angle in radians, to degrees, minutes, and seconds. The real number is read from the keyboard and has values between 0 and 2*PI. Use the following formulas: angle_degrees = angle_radians * 180/PI, degrees = integer part of angle_degrees, minutes = integer part of the expression ((angle_degrees - degrees) * 60), and seconds = (angle_degrees - degrees - (minutes / 60.0)) * 3600. The result should be displayed like this: degrees°minutes'seconds".

9. Write a C program to set the n^{th} bit of an unsigned integer number read from the keyboard. Use the following: left shift 1, n times, and then perform bitwise OR with the unsigned integer number.

10. Write a C program to verify if a year, read from the keyboard, is leap or not using the conditional operator. Use the following: if a year is divisible by 4 and not divisible by 100, then it is a leap year; otherwise, if the year is divisible by 400, it is also a leap year; otherwise, it is a common year.

2.4 References

1. Numeric limits, https://en.cppreference.com/w/c/types/limits, Accessed in September 2024.
2. C − Operators, https://www.tutorialspoint.com/cprogramming/c_operators.htm, Accessed in September 2024.
3. Paul Deitel, Harvey Deitel, *C How to Program*, 2022, Ninth edition, Pearson Education, ISBN: 978-0-13-739839-3.
4. C Operator Precedence, https://en.cppreference.com/w/c/language/operator_precedence, Accessed in September 2024.

Laboratory paper 3

Statements in C

3.1 Overview

- Presentation of statements in C
- Use of statements within simple C programs
- Work time: 4 hours

3.2 Theoretical Considerations

A statement is a directive issued to the computer that instructs it to perform a particular action.

A computer program consists of a sequence of statements. C has the following statements grouped into five categories:

- Labeled statements: case, default;
- Expression statements;
- Decision making (selection) statements: if (single-selection), if ... else (double-selection), if ... else if ... else, switch (multiple-selection);
- Loop (iteration) statements: for, while, do ... while;
- Jump statements: break, continue, goto, return.

3.2.1 Labeled statements

A statement can be preceded by a label. A label is a simple identifier followed by a colon. The syntax of this type of statements is:

identifier:

C has two labeled statements used in switch statements: **case** and **default**, but the programmers can define more according to the algorithms they are implementing.

3.2.2 Expression statements

An expression statement comprises an expression followed by a semicolon. The syntax of this type of statements is:

expression;

These statements are used as assignments or as function calls. Several expression statements are provided in Listing 3.1.

```c
#include <stdio.h>
int main() {
  int first_integer = 0, second_integer = 0, minimum = 0;

  printf("Enter two integers: ");
  scanf("%d %d", &first_integer, &second_integer);
  minimum = first_integer < second_integer ? first_integer : second_integer;
  printf("The minimum of %d and %d is %d.\n", first_integer, second_integer,
     minimum);
  return 0;
}
```

Listing 3.1: Program L3Ex1.c

3.2.3 Decision making (selection) statements

The C language provides the following types of decision making statements:

- **if** statement – evaluates an expression and executes one or more statements when the expression is true;
- **if ... else** statement – an if statement can be accompanied by an else statement, which executes when the expression evaluates to false;
- **if ... else if ... else** statement – allows the evaluation of multiple expressions and executes different blocks of code for more than two conditions;
- **switch** statement – allows an expression to be verified for equality against a list of constant values.

if statement

The **if** statement has the following syntax:

```
if (expression) {
   statement(s) to be executed if the expression is true;
}
```

This statement has the following effect [1]:

- The expression is evaluated.
- If the result of the evaluation is true, then the block of code inside the **if** statement is executed.
- If the result of the evaluation is false, then the first statement after the end of the **if** statement is executed.

if ... else statement

The **if ... else** statement has the following syntax:

```
if (expression) {
    statement(s) to be executed if the expression is true;
}
else {
    statement(s) to be executed if the expression is false;
}
```

This statement has the following effect [1]:

- The expression is evaluated.
- If the result of the evaluation is true, then the block of code inside the **if** is executed.
- If the result of the evaluation is false, then the block of code inside the **else** is executed.

if ... else if ... else statement

The **if ... else if ... else** statement has the following syntax:

```
if (expression1) {
   statement(s) to be executed if the expression1 is true;
}
else if (expression2) {
   statement(s) to be executed if the expression1 is false and the expression2 is true;
}
else if (expression3) {
   statement(s) to be executed if the expression1 and the expression2 are false and the expression3 is true;
}
....
else {
   statement(s) to be executed if all the expressions are false;
}
```

This statement has the following effect:

- Expression1 is evaluated.
- If the result of the evaluation is true, then the block of code inside the **if** is executed.
- If the result of the evaluation is false, then expression2 is evaluated.
- If the result of the evaluation is true, then the block of code inside the first **else if** is executed.
- If the result of the evaluation is false, then expression3 is evaluated.
- If the result of the evaluation is true, then the block of code inside the second **else if** is executed.
- ....
- If the result of the evaluation is false, then the block of code inside the **else** is executed.

Listing 3.2 presents the usage of the **if ... else** and **if ... else if ... else** statements.

```c
#include <stdio.h>
int main() {
  int grade = 0;

  printf("Enter the number of points obtained at the exam: ");
  scanf("%d", &grade);
  if((grade >= 0) && (grade <= 10)){
    if(grade >= 9)
      printf("Passed with A.\n");
    else if(grade >= 8)
      printf("Passed with B.\n");
    else if(grade >= 7)
      printf("Passed with C.\n");
    else if(grade >= 6)
      printf("Passed with D.\n");
    else if(grade >= 5)
      printf("Passed with E.\n");
    else
      printf("Failed.\n");
  }
  else {
    printf("You entered an invalid grade!");
  }
  return 0;
}
```

Listing 3.2: Program L3Ex2.c

switch statement

The **switch** statement can be used instead of the **if ... else if ... else** statement. It is often faster than nested **if ... else**.

The **switch** statement has the following syntax:

```
switch (expression) {
   case constant1:
     statement(s) to be executed if the result of the expression is equal to constant1;
     break; /* optional */
   case constant2:
     statement(s) to be executed if the result of the expression is equal to constant2;
     break; /* optional */
   ....
   default: /* optional */
     statement(s) to be executed if the result of the expression doesn't match any constant;
}
```

This statement has the following effect [1]:

- The expression is evaluated.
- The result of the evaluation is compared with constant1, constant2, ...
- If the result is equal to a constant, then the block of code after the corresponding **case** label is executed.
- If the result doesn't match any constant, then the block of code after the **default** label is executed.

The **default** branch is optional. If it is missing and the value of the expression doesn't match any constant, the **switch** statement has no effect. Also, the **break** statement is optional. If it is missing, all the statements that follow are executed, until either a **break** is met or the **switch** statement ends. The constants must be the same data type as the expression in the switch. The data types allowed are integer and character.

Listing 3.3 presents the usage of the **switch** statement.

```
#include <stdio.h>
int main() {
  int grade = 0;

  printf("Enter one of the following A, a, B, b, C, c, D, d, E, e, F, f: ");
  grade = getchar();
  switch(grade) {
    case 'a':
    case 'A': printf("Grade in the following interval [9, 10].\n");
      break;
    case 'b':
    case 'B': printf("Grade in the following interval [8, 9).\n");
      break;
    case 'c':
    case 'C': printf("Grade in the following interval [7, 8).\n");
      break;
    case 'd':
    case 'D': printf("Grade in the following interval [6, 7).\n");
      break;
    case 'e':
    case 'E': printf("Grade in the following interval [5, 6).\n");
      break;
    case 'f':
    case 'F': printf("Grade in the following interval [0 and 5).Failed.\n");
      break;
```

```
26      default: printf("You entered an invalid grade!");
27    }
28    return 0;
29  }
```

Listing 3.3: Program L3Ex3.c

3.2.4 Loop (iteration) statements

The C language provides the following types of loop statements:

- **for** statement – executes one or more statements multiple times using a loop variable;
- **while** statement – repeats one or more statements while a given expression is true and evaluates the expression before executing the loop body;
- **do ... while** statement – repeats one or more statements while a given expression is true and evaluates the expression at the end of the loop body.

for statement

The **for** statement has the following syntax:

```
for (expression1; expression2; expression3) {
   statement(s) to be executed;
}
```

This statement has the following effect:

- Expression1 is executed first and only a single time. This step allows programmers to define and initialize any loop control variables.
- Expression2 is evaluated.
- If the result of the evaluation is true, then the block of code inside the **for** is executed.
- If the result of the evaluation is false, then the first statement after the end of the **for** is executed.
- After the block of code inside the **for** executes, expression3 is executed. This step allows programmers to update any loop control variables.
- Expression2 is evaluated again. If it is true, the loop executes and the process continues to repeat. After the expression2 becomes false, the **for** ends.

expression1, expression2 and expression3 may be missing, but the presence of **;** is mandatory.

A loop turns into an infinite loop if the evaluation of expression2 is always true. Additionally, since none of the three expressions in the **for** statement are mandatory, an infinite loop can be created by leaving expression2 empty.

Listing 3.4 presents the usage of the **for** statement.

```
1  #include <stdio.h>
2  #define MAX_SIZE 100
3  int main() {
4    float numbers[MAX_SIZE] = {0}, average = 0.0f, sum = 0.0f;
5    int   i = 0, no_numbers = 0;
6
7    printf("Compute the arithmetic average value of n (<%d) real numbers.\n",
       MAX_SIZE);
8    printf("Input the number of real numbers, n = ");
9    scanf("%d", &no_numbers);
10   if(no_numbers > 0 && no_numbers <= MAX_SIZE) {
```

```
11      printf("Input the real numbers:\n");
12      for(i=0, sum=0; i < no_numbers; i++) {
13        printf("numbers[%3d] = ", i + 1);
14        scanf("%f", &numbers[i]);
15        sum += numbers[i];
16      }
17      average = sum / no_numbers;
18      printf("\nAVERAGE = %.2f\n", average);
19    }
20    else {
21      printf("The number of real numbers is invalid.");
22    }
23    return 0;
24  }
```

Listing 3.4: Program L3Ex4.c

while statement

The **while** statement has the following syntax:

```
while (expression) {
   statement(s) to be executed;
}
```

This statement has the following effect [1]:

- The expression is evaluated.
- If the result of the evaluation is true, then the block of code inside the **while** is executed and the expression is evaluated again.
- If the result of the evaluation is false, then the first statement after the end of the **while** is executed.

It is possible that a **while** statement might not execute at all. When the expression is evaluated as false from the beginning, the block of code within the **while** is skipped and the first statement after the **while** is executed. The block of code within a **while** statement has to contain some statements which change the value of the variables from the expression.

Listing 3.5 presents the usage of the **while** statement.

```
1  #include <stdio.h>
2  #define PERIOD '.'
3  int main() {
4    int character = 0, no_character = 0;
5
6    printf("Computes how many times the characters other than single or double
       quotes appear in an input sentence. Type . to end the sentence.\n");
7    printf("Input sentence: ");
8    while((character = getchar()) != PERIOD) {
9      if((character != '"') && (character != '\''))
10       no_character++;
11   }
12   printf("There are %d non quote characters.\n", no_character);
13   return 0;
14 }
```

Listing 3.5: Program L3Ex5.c

do ... while statement

The **do ... while** statement has the following syntax:

```
do {
   statement(s) to be executed;
} while (expression);
```

This statement has the following effect [1]:

- The block of code inside the **do ... while** is executed.
- The expression is evaluated.
- If the result of the evaluation is true, then the block of code inside the **do ... while** is executed and the expression is evaluated again.
- If the result of the evaluation is false, then the first statement after the end of the **do ... while** is executed.

Unlike **for** and **while** loop statements, which evaluate the expression at the beginning of the loop, the **do...while** statement verifies its expression at the end of the loop. This feature guarantees that **do...while** is executed at least one time.

Listing 3.6 presents the usage of the **do ... while** statement.

```
#include <stdio.h>
#define PERIOD '.'
int main() {
  int character = 0, no_character = 0;

  printf("Computes how many times the characters other than single or double
     quotes appear in an input sentence. Type . to end the sentence.\n");
  printf("Input sentence: ");
  do {
    if((character != '"') && (character != '\''))
      no_character ++;
  } while((character = getchar()) != PERIOD);
  printf("There are %d non quote characters.\n", no_character - 1);
  return 0;
}
```

Listing 3.6: Program L3Ex6.c

3.2.5 Jump statements

C language provides the following types of jump statements:

- **break** statement – terminates the loop or switch statements and transfers execution to the statement that follows the loop or switch;
- **continue** statement – causes the loop to skip the remaining statements in its body and immediately retests the loop expression before reiterating;
- **goto** statement – redirects the execution to a labeled statement;
- **return** statement – returns from a function.

break statement

The **break** statement has the following syntax:

```
break;
```

The **break** statement is used in a block of code inside loop statements or in the **switch** statement.

The **break** statement has the following effect: when it is encountered within a loop, the loop is immediately terminated, and the execution continues with the first statement following the loop.

Listing 3.7 presents the usage of the **break** statement.

```c
#include <stdio.h>
#define MAX_DIGIT 9
int main() {
  int sum = 0, digit = 0;

  printf("Computes the sum of the digits from 0 to %d.\n", MAX_DIGIT);
  while(1) {
    if(digit > MAX_DIGIT)
      break;
    sum += digit;
    digit++;
  }
  printf("The sum of the digits is %d.", sum);
  return 0;
}
```

Listing 3.7: Program L3Ex7.c

continue statement

The **continue** statement has the following syntax:

continue;

This statement is used only in a block of code inside the loop statements.

The **continue** statement has the following effect: when it is encountered within a loop, it forces the next iteration iteration to occur, bypassing any code in between.

Listing 3.8 presents the usage of the **continue** statement.

```c
#include <stdio.h>
#define MAX 3
int main() {
  double number = 0.0, sum = 0.0;
  int    i      = 0;

  printf("Computes sum of %d real numbers. Negative numbers are skipped from
     calculation.\n", MAX);
  for(i = 0; i < MAX; i++) {
    printf("Enter the n%d real number: ", i + 1);
    scanf("%lf", &number);
    if(number < 0)
      continue;
    sum += number;
  }
  printf("The sum of the positive numbers is %.2lf.\n", sum);
  return 0;
}
```

Listing 3.8: Program L3Ex8.c

goto statement

The **goto** statement has the following syntax:

```
goto label;
....
label: statement;
```

This statement enables an unconditional jump from the **goto** keyword to a labeled statement within the same function. The use of **goto** statement is strongly discouraged as it complicates the ability to follow the control flow of a program, making it harder to understand and modify.

return statement

The **return** statement has the following syntax:

```
return expression;
```

This statement ends the execution of a function and returns an expression from a function. The expression has the same data type as the return data type of the function. If the function return data type is void, the function does not return an expression.

3.3 Lab Tasks

1. Run the programs given as examples and analyze the output.
2. What will be the output of the program?

```c
#include<stdio.h>
int main() {
  int number = 100;

  switch(number) {
    case 100:
    printf("Case 1");
    case 200:
    printf("Case 2");
    break;
    case number:
    printf("Case 3");
    break;
  }
  return 0;
}
```

A. Case 1
B. Case 1Case 2
C. Error: no default value is specified
D. Error: case label does not reduce to an integer constant

3. What will be the output of the program?

```c
#include<stdio.h>
int main() {
  unsigned short int i = 0;

  for(i <= 10 && i >= -1; ++i; i > 0)
    printf("%u ", i);
  return 0;
}
```

A. 1 2 3 4 5 6 7 8 9 10 ... 65535
B. Expression syntax error
C. No output
D. 1 2 3 4 5 6 7 8 9 10

4. What does the following program print?

```c
#include<stdio.h>
int main() {
  int a = 1, b = 0;

  while(a <= 5) {
    b = 1;
    while(b <= a) {
      printf("%d", a);
      b = b + 1;
    }
    printf("\n");
    a = a + 1;
  }
  return 0;
}
```

5. What will be the output of the program?

```c
#include<stdio.h>
int main() {
  int a = 1;

  while(a <= 10){
    printf("%d ", a);
    if(a > 3)
      break;
    a++;
  }
  printf("%d", a + 10);
  return 0;
}
```

A. 1 2 3 4 14
B. 1 2 3 3 13
C. 1 2 3 3 14
D. 1 2 3 3 10

6. What will be the output of the program?

```c
#include<stdio.h>
int main() {
  int a = 1;

  while(a <= 10){
    printf("%d ", a);
    if(a > 3 && a < 8)
      continue;
    a++;
  }
  printf("%d", a + 10);
  return 0;
}
```

A. 1 2 3 4 5 6 ..... infinite
B. 1 2 3 4 5 15
C. 1 2 3 4 4 4 ..... infinite
D. 1 2 3 3 13

7. Write a C program which reads a real value for **n** and then computes the value for the function:

$$f(n) = \begin{cases} n^2 + 3n + 5 & \text{if } n < -3 \\ -3 & \text{if } n = -3 \\ n^2 - 10n + 2 & \text{if } n > -3 \end{cases}$$

8. Write a C program to display 100 times a special character inserted by the user from the keyboard. After every ten special characters, the program should output a newline character.

9. Write a C program to display an unsigned integer in binary format using bitwise operators. Use the bitwise AND operator to combine the value of the unsigned integer with the 1 « 31 mask, and also a shift operator to modify the value of the mask.

10. Write a C program that reads from the keyboard integer values until 0 is entered. Once the input is complete, the program calculates and displays the total count of even integers (excluding 0) entered, the average of the even integers, the total count of odd integers entered, and the average of the odd integers. Analyze the cases when only even or odd numbers are read from the keyboard, or when 0 is entered from the beginning.

11. Write a C program that presents a menu with options for addition (+), subtraction (-), multiplication (*) or division (/). After the user makes a selection, the program prompts for two real numbers and performs the chosen operation. The program should allow the user to repeat the operation until the 'q' key is pressed.

12. Write a C program to sort an array of integer values in descending order. Use the "bubble sort" sorting algorithm [2] (the algorithm repeatedly compares two adjacent elements of the array and swaps them if they are not arranged in descending order).

3.4 References

1. Brian Kernighan, Dennis Ritchie, *The C Programming Language*, 1988, Second edition, Prentice Hall, ISBN: 0-13-110370-9.
2. Soni Upadhyay, Bubble Sort Algorithm: Overview, Time Complexity, Pseudocode and More, https://www.simplilearn.com/tutorials/data-structure-tutorial/bubble-sort-algorithm, Accessed in September 2024.

Laboratory paper 4

Pointers in C

4.1 Overview

- Presentation of pointers in C
- Use of pointers within simple C programs
- Work time: 2 hours

4.2 Theoretical Considerations

A pointer is a derived data type in C that holds an address of a memory location. In other words, it points to the location of a block of memory that in turn contains actual data of interest.

A variable of type pointer holds the address of another variable. Therefore, a variable name *directly* references a value, while a pointer *indirectly* references a value [1]. The way of referencing a value through a pointer is known as *indirection*.

The purpose of pointers is to allow the programmers to manually, directly access a block of memory. Pointers are used predominately for *strings* and *structs*.

4.2.1 Pointer variable definition

The following syntax is used to define a pointer variable in C:

data_type *identifier;

where:

- *data_ type* is the type of the pointer (pointer to a data type);
- * is the symbol used to define a pointer (in this case * is not the dereference operator);
- *identifier* is the name of the pointer.

Several definitions of pointer variables are presented below.

```
1 int  *int_ptr;   //a pointer to an integer
2 char *char_ptr;  //a pointer to a character
```

4.2.2 Pointer variable initialization (assignment)

The following syntaxes are used to initialize a variable of type pointer or to assign a value to this type of variable:

identifier = value;
data_type *identifier = value;

where:

- *value* is the address (location in memory) of a variable. The reference operator (&) returns the address of a variable.

Several initializations (assignments) of pointer variables are presented below.

```
1 int var1     = 5;
2 int *int_ptr = NULL;
3
4 int_ptr = &var1;         //int_ptr takes the address of integer variable var1
5
6 char var2      = 'A';
7 char *char_ptr = &var2; //char_ptr takes the address of character variable
8                          //var2
```

4.2.3 Dereferencing a pointer

Accessing the value that a pointer points to is called *dereferencing a pointer*. The dereference operator (*) returns the value of the variable to which its operand points [2]. So, it is used to dereference a pointer. For example *ptr* is the address of a variable, whereas **ptr* is the value of the variable that *ptr* points to.

Listing 4.1 presents an example of dereferencing a pointer and also highlights the fact that & and * operators are complements of each other.

```c
#include <stdio.h>

int main () {
  int var1     = 1025;
  int *int_ptr = &var1;

  printf("Address of var1: %p\n", &var1);
  printf("Value of int_ptr: %p\n", int_ptr);
  printf("Address of int_ptr: %p\n", &int_ptr);
  printf("Value of var1: %d\n", *int_ptr);
  printf("Value of *int_ptr: %d\n", *int_ptr);
  printf("Value of &*int_ptr: %p and of *&int_ptr: %p\n", &*int_ptr, *&
    int_ptr);
  return 0;
}
```

Listing 4.1: Program L4Ex1.c

Figure 4.1 presents graphically what happens in the computer memory when the first two statements from Listing 4.1 are executed. The value of the pointer is represented in binary in computer memory using 8 bytes if a 64-bit operating system is used.

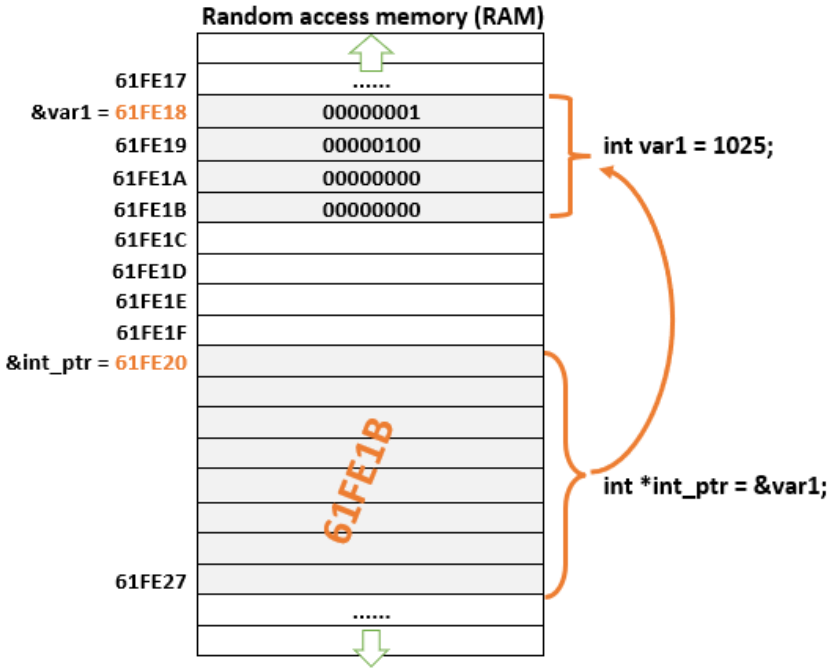


Figure 4.1: Memory representation of a pointer to integer definition and initialization (a pointer is represented on 8 bytes if a 64-bit machine is used, otherwise it requires only 4 bytes). var1 integer is represented in little endian format.

4.2.4 Pointer to void

A *pointer to void* is used when a pointer must not be bound to any data type. The main reason for using it is the re-usability of the pointer variable.

The following syntax is used to define a pointer to void in C:

void *identifier;

A pointer to void cannot be dereferenced. However, using the cast operator a pointer to void can be dereferenced (see the following syntax). Void pointers are of great use when dynamic allocation of the memory is performed.

***(data_type *)identifier**

An example of a pointer to void definition and initialization and the way in which it can be dereferenced are presented below.

```
int   integer_var;
float float_var;
void  *ptr = NULL;

ptr = &integer_var;
ptr = &float_var;
*ptr = 7.0f; //Error -- ptr is a pointer to void and it cannot be
             //dereferenced
*(float *)ptr = 7.0f; //Correct -- a type cast is required
```

4.2.5 Constant pointers

The *const* keyword informs the compiler that the value of a variable should not be modified. Using *const*, two types of pointer variables can be defined: pointer to a constant value and constant pointer. These pointers must be initialized when they are defined.

A *pointer to a constant* value can be defined using the following syntaxes:

data_type const *identifier = value;
const data_type *identifier = value;

A pointer to a constant value can be changed to point to any value of the appropriate data type, but the value it points to cannot be modified [2]. For example, a pointer to a constant value can be used as an array argument of a function that will process each array's element without modifying its value.

```
int num1 = 20, num2 = 5;
int const *ptr = &num1; //Definition and initialization of a pointer to
                        //a constant integer value (read-only *ptr)
*ptr = 20;              //Error - *ptr is const (cannot modify a const)
ptr++; ptr = &num2;     //Correct - ptr is not const
```

A *constant pointer* can be defined using the following syntax:

data_type *const identifier = value;

A constant pointer always points to the same memory location, and the value at that location can be changed through the pointer [2]. An array name is a constant pointer to the beginning of the array. All the values of the array's element can be accessed and modified

by using the array name along with array subscripting. For example, a constant pointer can be passed as an array argument to a function that accesses the array's elements using only the array subscript notation.

```
int num1 = 20, num2 = 5;
int *const ptr = &num1; //Definition and initialization of a constant
                        //pointer to an integer (read-only variable ptr)
*ptr = 20;              //Correct - *ptr is not const
ptr++; ptr = &num2;     //Error - ptr is const (cannot assign new address)
```

4.2.6 Pointer arithmetic

The pointer variables can be used as operands in arithmetic, assignment and comparison expressions.

The next set of arithmetic and/or assignment operations may be performed [2]:

- An integer may be added to a pointer (+ or +=);
- An integer may be subtracted from a pointer (- or -=);
- A pointer may be incremented (++) or decremented (−−);
- One pointer may be subtracted from another.

Pointer variables can be compared using the C relational operators (==, !=, <, <=, >, >=). Comparisons of pointer variables evaluate the addresses contained within those pointer variables. A typical application of comparing pointer variables is to check if a pointer is NULL.

When an integer n is added/subtracted to/from a pointer *ptr*, the value of the pointer is increased/decreased with n * the number of bytes required to store a data item of the data type of the pointer. The following formulas are used:

$ptr + n$ equivalent to $ptr + n * sizeof(ptr\_data\_type)$
$ptr+ = n$ equivalent to $ptr = ptr + n * sizeof(ptr\_data\_type)$
$ptr - n$ equivalent to $ptr - n * sizeof(ptr\_data\_type)$
$ptr- = n$ equivalent to $ptr = ptr - n * sizeof(ptr\_data\_type)$

When a pointer is incremented/decremented by one, the value of the pointer is increased/decreased with the number of bytes required to store a data item of the data type of the pointer. The following formulas are used:

$ptr++$ or $++ptr$ equivalent to $ptr + sizeof(ptr\_data\_type)$
$ptr--$ or $--ptr$ equivalent to $ptr - sizeof(ptr\_data\_type)$

When one pointer *ptr2* is subtracted from another pointer *ptr1*, the result is the number of bytes between the two addresses.

$no = ptr1 - ptr2$, where $ptr1 > ptr2$

The expression $*++ptr$ increments *ptr* to point to the next address and then retrieves the value from the updated address.

The expression $++*ptr$ takes the value indirectly referenced by *ptr* and increments it.

Pointer arithmetic is not meaningful unless it is applied to an array. It cannot be presumed that two or more variables of the same data type are stored contiguously in memory unless they are consecutive elements of an array.

4.2.7 Pointers and arrays

An array name can be considered a constant pointer to its first element and cannot be changed during execution. Figure 4.2 presents the memory representation of an array of integers definition and initialization.

Pointers can be used to carry out any operation that involves array subscripting. Listing 4.2 exemplifies four methods to refer to one dimensional array elements: array subscripting, array offset, pointer subscripting, and pointer offset.

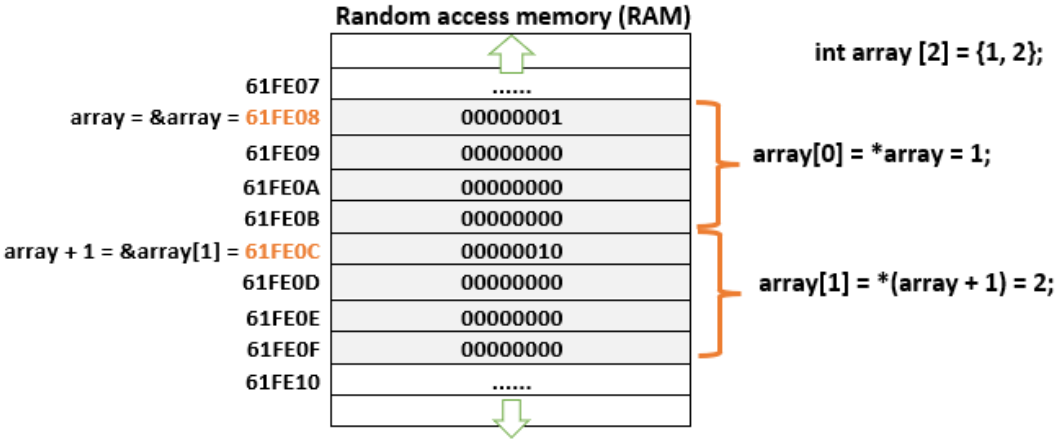


Figure 4.2: Memory representation of an array of integers definition and initialization

```c
#include <stdio.h>

int main() {
  int array[5]   = {1, 2, 3, 4, 5}; /* define and initialize array */
  int *array_ptr = array;           /* define and initialize array_ptr to
                                       point to the first array element */
  /* int *array_ptr = &array[0]; equivalent to the definition from the
    previous line */
  int i      = 0;
  int offset = 0;

  /* output array using array subscript notation */
  printf("Array printed with:\nArray subscript notation\n");
  for (i = 0; i < 5; i++) {
    printf("array[%d] = %d\n", i, array[i]);
  }

  /* output array using array offset notation */
  printf("\nArray offset notation\n");
  for (offset = 0; offset < 5; offset++) {
    printf("*(array + %d ) = %d\n", offset, *(array + offset));
  }

  /* output array using pointer subscript notation */
  printf("\nPointer subscript notation\n");
  for (i = 0; i < 5; i++) {
    printf("array_ptr[%d] = %d\n", i, array_ptr[i]);
  }

  /* output array using pointer offset notation */
  printf("\nPointer offset notation\n");
```

```
31   for (offset = 0; offset < 5; offset++) {
32     printf("*(array_ptr + %d ) = %d\n", offset, *(array_ptr + offset));
33   }
34   return 0;
35 }
```

Listing 4.2: Program L4Ex2.c

Additionally, the previous example demonstrates the definition and use of a pointer which points to the first element of an array (array_ptr). Furthermore, a pointer that points to the whole array can be defined using the following syntax [3]:

data_type (*identifier)[integer_constant];

where:

- *data_type* is the data type of the pointer;
- *identifier* is the name of the pointer to an array;
- *integer_constant* is the size of the array to which the pointer points to.

This type of pointer is useful when multidimensional arrays are used. Listing 4.3 shows the difference between a pointer that points to the first element of an array and a pointer that points to the entire array.

```
1  #include<stdio.h>
2
3  int main() {
4    int array[2][5] = {{1, 2, 3, 4, 5},
5                       {6, 7, 8, 9, 10}};  /* define and initialize array */
6    int (*ptr)[5]   = array; /* define and initialize ptr as a pointer to
7                                array */
8    int *p          = array; /* define and initialize p to point to the first
9                                element of array */
10
11   printf("p = %p, ptr = %p\n", p, ptr);
12   printf("array[0][0]=%d, array[0]=%p, array[0][0]=%d\n", *p, *ptr, **ptr);
13   p++;    /* p = p + 4 bytes (the size of an element of the array)
14              -> points to the array[0][1] element of the array */
15   ptr++;  /* ptr = ptr + 20 bytes (the entire size of array's row)
16              -> points to the next row */
17   printf("p = %p, ptr = %p\n", p, ptr);
18   printf("array[0][1]=%d, array[1]=%p, array[1][0]=%d\n", *p, *ptr, **ptr);
19   return 0;
20 }
```

Listing 4.3: Program L4Ex3.c

4.2.8 Array of pointers

An array of pointers stores addresses of more than one variable into a single pointer [3]. The following syntax is used to define an array of pointers in C:

data_type *identifier[integer_constant];

where:

- *data_type* is the data type of the pointers within the array;
- *identifier* is the name of the array of pointers;
- *integer_constant* is the size of the array of pointers.

A typical application of an array of pointers is to create an array of strings (a string is an array of characters). Figure 4.3 presents the memory representation of an array of two strings definition and initialization. An array of pointers containing two pointers, each of them pointing to a character, is used. Each pointer of the array points to the first character of a string. The value of the pointer is represented in binary using 8 bytes if a 64-bit machine is used. Each character is represented in binary using the ASCII (American Standard Code for Information Interchange) Code (i.e., 'c' -> 01100011).

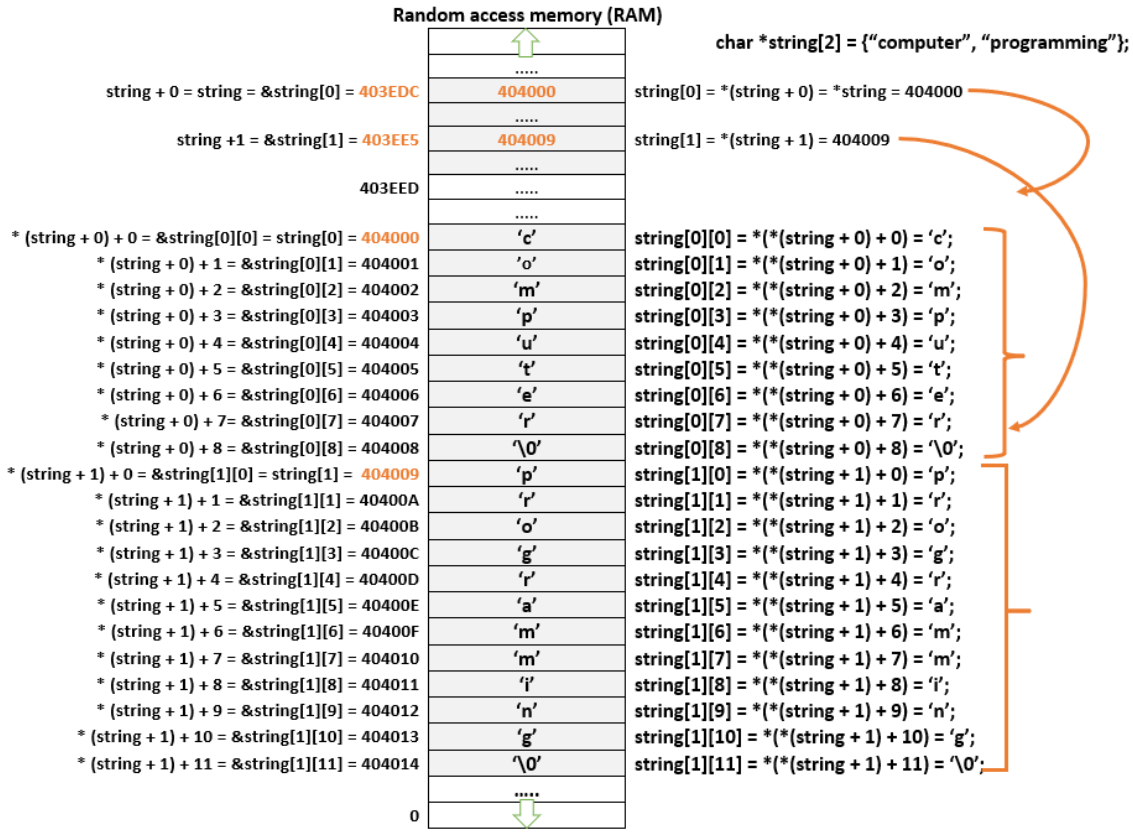


Figure 4.3: Memory representation of an array of two arrays of characters (strings) definition and initialization

Listing 4.4 and Listing 4.5 show methods to refer to array elements: array subscripting, and respectively array offset. These notations can be used also when dealing with two dimensional arrays of any data type.

```
#include <stdio.h>

int main() {
  char *string[2] = {"computer", "programming"};
  int  i          = 0;
  int  j          = 0;

  printf("&string[0] = %p.\n", &string[0]);
  printf("&string[1] = %p.\n", &string[1]);
  printf("string[0] = %p.\n", string[0]);
  printf("string[1] = %p.\n", string[1]);

  /* output array using array subscript notation */
  printf("Pointer subscript notation:\n");
```

```
15   for(i = 0; i < 2; i++){
16     printf("String started at string[%d] is %s.\n", i, string[i]);
17     for(j = 0; j < strlen(string[i]); j++)
18       printf("string[%d][%d] = %c at address &string[%d][%d] = %p.\n", i, j,
       string[i][j], i, j, &string[i][j]);
19   }
20   return 0;
21 }
```

Listing 4.4: Program L4Ex4.c

```
1  #include <stdio.h>
2
3  int main() {
4    char *string[2] = {"computer", "programming"};
5    int  i          = 0;
6    int  j          = 0;
7
8    printf("string + 0 = %p.\n", string + 0);
9    printf("string + 1 = %p.\n", string + 1);
10   printf("*(string + 0) = %p.\n", *(string + 0));
11   printf("*(string + 1) = %p.\n", *(string + 1));
12
13   /* output array using array offset notation */
14   printf("\nPointer offset notation\n");
15   for (i = 0; i < 2; i++){
16     printf("String started at *(string + %d) is %s.\n", i, *(string + i));
17     for(j = 0; j < strlen(*(string + i)); j++)
18       printf("*(*(string + %d) + %d) is %c at address *(string + %d) + %d =
     %p.\n", i, j, *(*(string + i) + j), i, j, *(string + i) + j);
19   }
20   return 0;
21 }
```

Listing 4.5: Program L4Ex5.c

4.2.9 Pointer to pointer

A pointer to a pointer can be used for multiple indirection or to create a chain of pointers. When a pointer to a pointer is defined, the first pointer holds the address of the second pointer, which in turn points to the location containing the actual value. The following syntax is used to define pointer to a pointer in C:

data_type **identifier;

Listing 4.6 presents a simple example of using a pointer to a pointer in C.

```
1  #include <stdio.h>
2
3  int main() {
4    int number    = 5;
5    int *ptr      = &number;
6    int **ptr_ptr = &ptr;
7
8    printf("Address of ptr_ptr: %p\n", &ptr_ptr);
9    printf("Value of ptr_ptr: %p\n", ptr_ptr);
10   printf("Address of ptr: %p\n", &ptr);
11   printf("Value of ptr: %p\n", ptr);
12   printf("Address of number: %p\n", &number);
```

```
13   printf("Value of number: %d\n", **ptr_ptr);
14   printf("Value of number: %d\n", *ptr);
15   printf("Value of number: %d\n", number);
16   return 0;
17 }
```

Listing 4.6: Program L4Ex6.c

4.3 Lab Tasks

1. Run the programs given as examples and analyze the output.

2. Identify the error in each of the following sequences of code assuming the next definitions:

```
int  array[5]  = {1, 2, 3, 4, 5};
void *void_ptr = array;
int  number    = 0
int  i         = 0;
int  *array_ptr;
```

   a) ++array_ptr;
   b) /* use array_ptr to get the first value of array */
   number = array_ptr;
   c) /* assign array element 3 (the value 4) to number */
   number = *array_ptr[3];
   d) /* print all the elements of array */
   for(i = 0; i <= 5; i++) printf("%d ", array_ptr[i]);
   e) /* assign the value pointed to by void_ptr to number */
   number = *void_ptr;
   f) ++array;

3. For each of the following, write a single statement to accomplish the indicated task. Assume that x and y are variables of type double that have been defined, and that x has been initialized to 10.00.
   a) Define the variable *double_ptr* to be a pointer to a double.
   b) Assign the address of variable x to pointer variable *double_ptr*.
   c) Display the value that is pointed to by *double_ptr*.
   d) Assign the value pointed to by *double_ptr* to variable y.
   e) Display the value held by variable y.
   f) Display the address of variable x. Use the %p format specifier.
   g) Display the address stored in *double_ptr*. Use the %p format specifier. Is the displayed value the same as the address of variable x?

4. Write a C program to display the reverse of a string (array of characters) using pointers.

5. Write a C program to sort an array of integers in descending order using a pointer that points to the first element of the array and the pointer offset notation. Use the "bubble sort" sorting algorithm [4] (the algorithm repeatedly compares two adjacent elements of the array and swaps them if they are not arranged in descending order).

4.4 References

1. Paul Deitel, Harvey Deitel, *C How to Program*, 2022, Ninth edition, Pearson Education, ISBN: 978-0-13-739839-3.
2. Richard M. Reese, Understanding and Using C Pointers, 2013, O'Reilly Media, Inc., ISBN: 978-1-44-934418-4.
3. Pointers in C Explained – They're Not as Difficult as You Think, https://www.freecodecamp.org/news/pointers-in-c-are-not-as-difficult-as-you-think/, Accessed in September 2024.
4. Soni Upadhyay, Bubble Sort Algorithm: Overview, Time Complexity, Pseudocode and More, https://www.simplilearn.com/tutorials/data-structure-tutorial/bubble-sort-algorithm, Accessed in September 2024.

Laboratory paper 5

Functions in C

5.1 Overview

- Presentation of functions in C
- Use of functions within simple C programs
- Work time: 4 hours

5.2 Theoretical Considerations

A function is a component of a C program designed to perform a specific task.

Functions allow the programmers to modularize a program, to manage its complexity, and also assure software reusability.

C programs are usually written by combining *programmer-defined functions* (functions defined by the programmers) with *built-in functions* available in the C Standard Library. Each C program has to contain the *main* function, which represents its entry-point.

Functions are invoked through function calls which specify the functions name and provide the necessary information (as arguments) that the called functions require to execute their tasks.

5.2.1 Function definition

The following syntax is used to define a function in C:

```
return_value_data_type identifier(formal parameters list) {
  //body of the function
  local variables definition and initialization;
  statement(s);
}
```

where:

- *return_value_data_type* is the data type of the result (default int);
- *identifier* is the name of the function;
- *formal parameters list* may contain:
  - No parameters (syntax: **return_value_data_type identifier() {...}** or syntax: **return_value_data_type identifier(void) {...}**)
  - One or more parameters, separated by commas and specified by: *data_type formal_parameter_identifier*.

A formal parameter may be preceded by the *const* keyword. In this case its value becomes constant in the function body (the parameter's value is read only). Accordingly, any attempt to alter the value of that parameter in the function body generates an error.

A function may contain arrays as its formal parameters.

If a formal parameter is an one dimensional array, it can be defined as:

- *data_type formal_parameter_identifier[ ]*
- *data_type *formal_parameter_identifier*

The two forms are equivalent. The subscript notation, *formal_parameter_identifier[index]*, can be used in the body of the function.

If a formal parameter is a two dimensional array, it can be defined as:

- *data_type formal_parameter_identifier[ ][no_columns]*
- *data_type (*formal_parameter_identifier)[no_columns]*

The two forms are equivalent. The following subscript notation can be used in the body of the function, *formal_parameter_identifier[row][column]*.

When an array formal parameter is prefixed with the *const* keyword, the values of the array elements are treated as constants within the function body. Accordingly, any attempt to change the value of an element of the array in the function body generates an error.

Unlike char data type arrays, other types of array do not have a special terminator. Therefore, it is recommended that the size of the array to be stated as a formal parameter, so that the function processes the correct number of elements.

There are two categories of functions in C: functions which return a value and functions which do not return a value. When a function returns a value, the last statement of the body of the function is *return expression;*, where *expression* has the same data type as the *return_value_data_type* of the function. When a function does not return a value, *return_value_data_type* is replaced with *void* and the last statement from the body of the function is *return;* which is optional because the function-ending right brace replaces it.

Several functions definitions are presented below:

```
/* square function definition - returns the square of its parameter */
int square(int y) {
  return y * y; /* returns square of y as an int */
}

/* cube_reference function definition - does not return a value */
void cube_reference(double *number_ptr){
  *number_ptr = pow(*number_ptr, 3);
}

/* print_mult_array function definition - does not return a value */
void print_mult_array(int size, int *ptr, int value) {
  int i = 0;
  for(i = 0; i < size; i++) {
    printf("\t %d", *ptr * value);
    ptr++;
  }
}

/* print_matrix function definition - does not return a value */
void print_matrix(int rows, int columns, const int (*ptr)[columns]) {
  int i = 0, j = 0;
  for(i = 0; i < rows; i++) {
    for(j = 0; j < columns; j++)
      printf("\t %d", ptr[i][j]);
    printf("\n");
  }
}
```

5.2.2 Function declaration

A function declaration informs the compiler about the data type that the function returns, as well as the data types of its parameters provided in the order they are expected. The compiler uses functions declarations to validate functions calls if the functions definitions are placed after their calls or when they are located in different source files.

The function prototype is used to declare a function as the following syntax shows:

return_value_data_type identifier(formal parameters list);

It is a good programming practice to specify function prototypes for all the required functions at the beginning of a source file or to include a header file containing these prototypes.

The declarations of the previous defined functions follow:

```
int  square(int y);                          //square function prototype
void cube_reference(double*);                //cube_reference function prototype
void print_mult_array(int, int*, int); //print_mult_array function prototype
void print_matrix(int, int, const int (*)[COLUMNS]); //print_matrix function
                                                     //prototype
```

The formal parameter names (not their data types!) can be missing.

5.2.3 Function call

A function has to be called (called function) by another function (caller) to perform its defined task. The following syntax is used to call a function which returns a value:

variable_identifier = identifier(effective parameters list);
identifier(effective parameters list);

The following syntax is used to call a function which does not return a value:

identifier(effective parameters list);

In both the above cases, the *effective parameters* (arguments) replace the *formal parameters*. The correspondence between formal and effective parameters is positional.

It is recommended that the data types of the formal parameters and of the effective parameters coincide. If this requirement is not satisfied, in C, the data type of the effective parameter is automatically converted to the data type of its corresponding formal parameter.

Several examples of the previous defined functions calls are presented below:

```
printf("%d", square(3));              //square function calls
int_variable = square(4);

cube_reference(&number);              //cube_reference function call

print_mult_array(SIZE, array, 2);     //print_mult_array function call

print_matrix(ROWS, COLUMNS, matrix); //print_matrix function call
```

The passing of the effective parameters can be done:

- By **value** (call by value);
- By **reference** (call by reference).

Call by value

When effective parameters are passed by value, copies of the arguments' values are created and passed to the called function [1]. The changes to the copies do not impact the original values of the effective parameters in the caller. *Call by value* should be used whenever the called function does not need to modify the original values of the caller's effective parameters.

Listing 5.1 presents the definition of a function that cubes a real number and its call using call by value.

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

double cube_value(double); /* cube_value function prototype */

int main() {
  double number = 5.0;

  printf("The original value of number is %.2lf.\n", number);
  number = cube_value(number);  /* pass number by value to cube_value */
  printf("The new value of number is %.2lf.\n", number );
  return 0;
}

/* calculate and return cube of a double */
double cube_value(double number) {
  return pow(number, 3);
}
```

Listing 5.1: Program L5Ex1.c

Call by reference

When effective parameters are passed by reference, the caller allows the called function to alter the original values of the effective parameters [1]. *Call by reference* should be used only with trusted called functions that need to modify the original values of caller's effective parameters.

Call by reference is not supported in C, but it's possible to simulate it by using reference and dereference operators (call by value using pointers).

Listing 5.2 presents the definition of a function that cubes a real number and its call using call by value through pointers.

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

void cube_reference(double*); /* cube_reference function prototype */

int main() {
  double number = 5.0;

  printf("The original value of number is %.2lf.\n", number);
  cube_reference(&number); /* pass number by value using pointer to
                              cube_reference */
  printf("The new value of number is %.2lf.\n", number );
  return 0;
}

/* calculate and return cube of a double */
void cube_reference(double *number_ptr) {
  *number_ptr = pow(*number_ptr, 3);
}
```

Listing 5.2: Program L5Ex2.c

Figure 5.1 and Figure 5.2 present graphically the programs from Listing 5.1 and Listing 5.2, respectively.

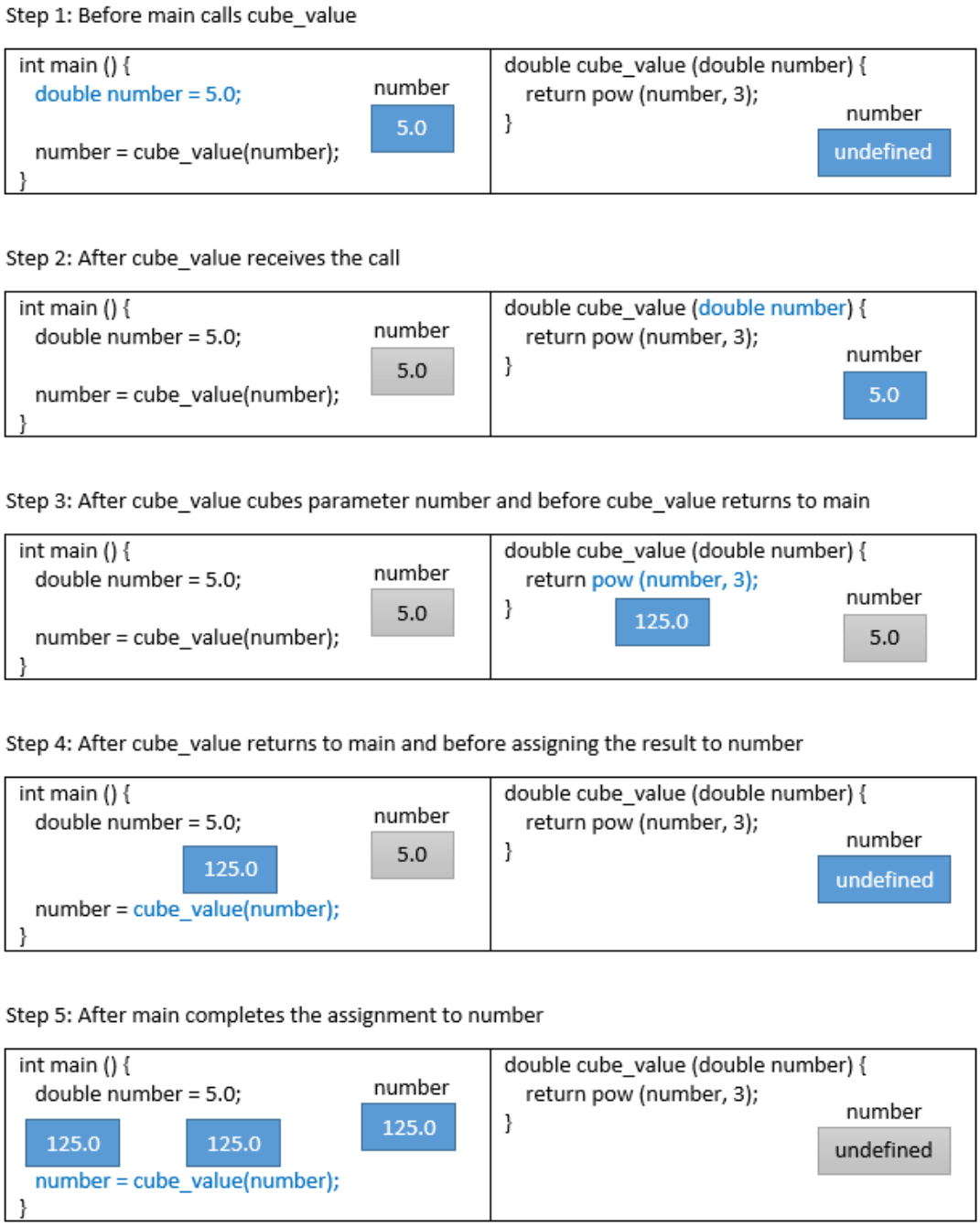


Figure 5.1: Analysis of Listing 5.1 – call by value (adapted from [1])

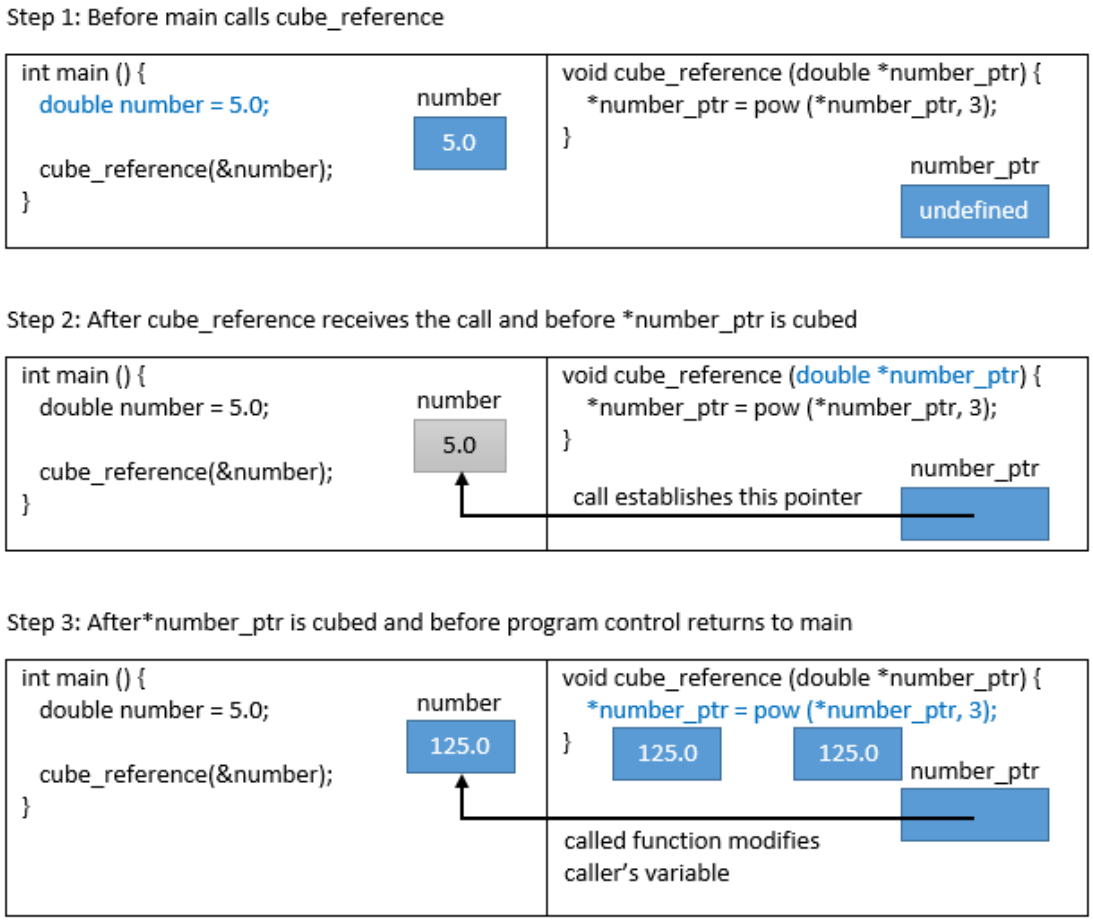


Figure 5.2: Analysis of Listing 5.2 – call by reference (adapted from [1])

Passing arrays to functions

The C programming language passes arrays to functions using call by reference (the called functions can modify the original elements values of the callers' arrays), using the name of the array (without any brackets) [1]. The name of the array evaluates to the address of the first element of the array.

Even if entire arrays are passed by reference, individual elements of an array are passed using call by value exactly as scalar variables are [1]. To pass an array element to a function, the subscripted name of the array element is used as an effective parameter (argument) in the function call.

Listing 5.3 exemplifies how to pass arrays to functions.

```
1  #include <stdio.h>
2  #include <math.h>
3  #define SIZE 3
4  #define ROWS 3
5  #define COLUMNS 3
6
7  void print_element(int);
8  void print_mult_array(int, int*, int);
9  void print_matrix(int, int, const int (*)[COLUMNS]);
10
```

```c
int main(){
  int array[SIZE]            = {1, 2, 3};
  int matrix[ROWS][COLUMNS] = {1, 2, 3, 4, 5, 6, 7, 8, 9};

  printf("The last element of the array: ");
  print_element(array[SIZE - 1]);      /* pass the last array element to
                                          print_element */
  printf("The array elements multiplied by 2: \n");
  print_mult_array(SIZE, array, 2);    /* pass array to print_multi_array */
  printf("\nThe matrix elements:\n");
  print_matrix(ROWS, COLUMNS, matrix);/* pass matrix to print_matrix */
  return 0;
}

/* print the value of an element of an array */
void print_element(int element) {
  printf("%d\n", element);
}

/* print the values of the elements of an array multiplied by a value */
void print_mult_array(int size, int *ptr, int value) {
  int i = 0;
  for(i = 0; i < size; i++) {
    printf("\t %d", *ptr * value);
    ptr++;
  }
}

/* print the values of the elements of a matrix */
void print_matrix(int rows, int columns, const int (*ptr)[columns]) {
  int i = 0, j = 0;
  for(i = 0; i < rows; i++) {
    for(j = 0; j < columns; j++)
      printf("\t %d", ptr[i][j]);
    printf("\n");
  }
}
```

Listing 5.3: Program L5Ex3.c

5.2.4 Function pointers

A pointer to a function holds the starting address in memory of the code that carries out the function's task. The function name represents this address.

The following syntax is used to define a pointer to a function in C:

return_value_data_type (*identifier)(formal parameters list);

A pointer to a function can be used to call the function directly as the function name or indirectly using the dereference operator.

Listing 5.4 presents the usage of a function pointer.

```c
#include<stdio.h>

void print_int(int);   /* print_int function prototype */

int main() {
  void (*f_ptr)(int);  /* define f_ptr a pointer to a function */

```

```c
  f_ptr = &print_int;  /* initialize f_ptr with the address of print_int */

  /* call print_int using the pointer to it */
  f_ptr(2);
  /* another way to call print_int using the pointer to it */
  (*f_ptr)(2);
  return 0;
}

/* print an integer */
void print_int(int x) {
  printf("%d\n", x);
}
```

Listing 5.4: Program L5Ex4.c

Pointers to functions can be stored in arrays, assigned to other function pointers, passed to other functions and returned from functions. A common application of function pointers is in text-based, menu-driven programs [1], as Listing 5.5 exemplifies.

```c
#include<stdio.h>

void fun_x(int x);   /* fun_x function prototype */
void fun_y(int y);   /* fun_y function prototype */
void fun_z(int z);   /* fun_z function prototype */

int main() {
  /* define and initialize f_ptr an array of 3 pointers to functions that
     each takes an integer argument and returns void */
  void(*f_ptr[3])(int) = {fun_x, fun_y, fun_z};
  int ch = 0;   /* define and initialize integer variable ch to hold
                   the user's choice */

  printf("Enter a number between 1 and 3 or other integer value to end: ");
  scanf("%d", &ch);

  while(ch >= 1 && ch < 4) {
    /* call function at location given by ch in array f_ptr and pass ch
       as an argument */
    (*f_ptr[ch - 1])(ch);
    printf("\nEnter a number between 1 and 3 or other integer value to end:
    ");
    scanf("%d", &ch);
  }
  return 0;
}

void fun_x(int x){
  printf("You entered %d --- fun_x was called.\n", x);
}

void fun_y(int y){
  printf("You entered %d --- fun_y was called.\n", y);
}

void fun_z(int z){
  printf("You entered %d --- fun_z was called.\n", z);
}
```

Listing 5.5: Program L5Ex5.c

5.2.5 Recursion

A recursive function is a function that calls itself either directly (i.e., contains one or more calls to itself) or indirectly through another function (i.e., contains a call to another function which, in turn calls the recursive function). An exit condition (base case) from the function has to be defined, otherwise an infinite loop results. To avoid infinite recursion, the recursive function has to be carefully constructed to ensure that at a certain time the function terminates without calling itself.

Several steps have to be considered when implementing recursive functions:

- Define a base case – identify the simplest case for which the solution is known or very simple. This represents the exit condition for recursion and ensures the ending of the function at a certain time.
- Define a recursive case – split the problem down into smaller versions of itself (subproblems), and call the function recursively to solve each of them.
- Ensure the recursion terminates – check that the recursive function, in the end, reaches the base case, and no infinite loop occurs.
- Merge the solutions – merge the solutions of the subproblems to solve the original problem.

Direct recursion

The following syntax is used to define direct recursion in C:

```
return_value_data_type identifier(formal parameters list) {
  //body of the function
  ...
  identifier(effective parameters list);
  ...
}
```

There are three types of direct recursion:

- Linear – occurs when an action has a simple repetitive structure consisting of some basic step followed by the action again (contains a call to itself) (i.e., used to calculate the factorial of a number)
- Binary – contains two calls to itself (i.e., used to generate the Fibonacci series)
- N-ary – represents the most general form of recursion, where N is not a constant, it is a parameter of the function (i.e., used to generate combinatorial objects such as permutations).

Listing 5.6 presents an example of direct recursion (linear).

```
1  #include <stdio.h>
2
3  unsigned int factorial(unsigned int); /* factorial function prototype */
4
5  int main() {
6    unsigned int i = 0;
7
8    printf("Enter an unsigned integer to calculate its factorial: ");
9    scanf("%u", &i);
10   /* call factorial function */
11   printf("Factorial of %u is %u.\n", i, factorial(i));
12   return 0;
13 }
```

```c
/* calculate the factorial of an unsigned integer using recursion */
unsigned int factorial(unsigned int i) {
  if(i <= 1)        /* base case or exit condition */
    return 1;
  else
    return i * factorial(i - 1);
}
```

Listing 5.6: Program L5Ex6.c

Indirect recursion

The following syntax is used to define indirect recursion in C:

```
return_value_data_type identifier1(formal parameters list) {
  //body of the function
  ...
  identifier2(effective parameters list);
  ...
}
return_value_data_type identifier2(formal parameters list) {
  //body of the function
  ...
  identifier1(effective parameters list);
  ...
}
```

Listing 5.7 presents an example of indirect recursion.

```c
#include <stdio.h>

unsigned int is_even(unsigned int); /* is_even function prototype */
unsigned int is_odd(unsigned int);  /* is_odd function prototype */

int main() {
  unsigned int i = 0;

  printf("Enter an unsigned integer to check if it is even or odd: ");
  scanf("%u", &i);
  if (is_even(i))   /* call is_even function */
    printf("%u is even.", i);
  else
    printf("%u is odd.", i);
  return 0;
}

/* check if an unsigned integer is even */
unsigned int is_even(unsigned int i) {
  if(i == 0)       /* base case */
    return 1;
  else
    return is_odd(i - 1);
}

/* check if an unsigned integer is odd */
unsigned int is_odd(unsigned int i) {
  if(i == 0)       /* base case */
```

```
29      return 0;
30    else
31      return is_even(i - 1);
32  }
```

Listing 5.7: Program L5Ex7.c

5.2.6 The C Standard Library

The C Standard Library offers numerous built-in functions prototypes, macros, and types for performing mathematical calculations, string and character manipulations, input/output operations, and other helpful tasks. To use them, the following headers (available in C89/C99/C11/C17/C23) have to be included in the program (Table 5.1).

Table 5.1: C Standard Library headers ([2])

Header	Explanation
<assert.h>	Contains macros and information for adding diagnostics that aid in program debugging.
<complex.h>	Contains macros and function prototypes for complex number arithmetic.
<ctype.h>	Contains function prototypes that test characters for certain properties, and function prototypes that can be used to convert lowercase letters to uppercase letters and vice versa.
<errno.h>	Defines macros that are useful for reporting error conditions.
<fenv.h>	Contains types, function prototypes and macros to support the floating-point environment.
<float.h>	Contains the floating-point size limits of the system.
<inttypes.h>	Provides integer types definitions and macros that are consistent across machines and independent of operating systems.
<limits.h>	Contains the integral size limits of the system.
<locale.h>	Contains function prototypes and other information that enable a program to be modified for the current locale on which it's running. The notion of locale enables the computer system to handle different conventions for expressing data like dates, times, dollar amounts and large numbers throughout the world.
<math.h>	Contains function prototypes for computing mathematical operations.
<setjmp.h>	Contains function prototypes for functions that allow bypassing of the usual function call and return sequence.
<signal.h>	Contains function prototypes and macros to handle various conditions that may arise during program execution.
<stdarg.h>	Defines macros for dealing with a list of arguments to a function whose number and types are unknown.
<stdatomic.h>	Contains operations on atomic types, macros and function prototypes for performing atomic operations on data shared between threads.
<stdbit.h>	Contains macros to work with the byte and bit representations of types.
<stdbool.h>	Contains macros for boolean type.
<stdckdint.h>	Contains macros for performing checked integer arithmetic.
<stddef.h>	Contains common type definitions used by C for performing calculations.
<stdint.h>	Contains types and macros used to specify fixed-width integer types.
<stdio.h>	Contains function prototypes for the standard input/output library functions, and information used by them.
<stdlib.h>	Contains function prototypes for conversions of numbers to text and text to numbers, memory allocation, random numbers, and other utility functions.
<stdnoreturn.h>	Contains the definition of the noreturn macro.
<string.h>	Contains function prototypes for string-processing functions.
<thmath.h>	Contains many types-generic macros used for mathematical operations. It includes <math.h> and <complex.h>.
<threads.h>	Contains macro, types and function prototypes that deal with threads management.
<time.h>	Contains function prototypes and types for manipulating the time and date.
<uchar.h>	Contains types and macros associated with extended character data types.
<wchar.h>	Contains types, macros and function prototypes to work with wide streams or to manipulate wide strings.
<wctype.h>	Contains macros and function prototypes to classify and map wide characters.

5.3 Lab Tasks

1. Run the programs given as examples and analyze the output.

2. Identify and fix the errors in the following function:

```
function(int x, int y) {
  int x;
  x = 20;
  return x;
}
```

3. What will be the output of the program?

```
#include <stdio.h>
void function(){
  static int s = 9;
  ++s;
  printf("%d ", s);
}
int main(){
  function();
  function();
  printf("%d ", s);
  return 0;
}
```

   A. 10 10 10
   B. 10 11 12
   C. 10 11 11
   D. error

4. What will be the output of the program?

```
#include <stdio.h>
int main() {
  int array[] = {10, 20, 30, 40, 50};
  int i, *array_ptr;

  array_ptr = array;
  for(i = 0; i < 4; i++) {
    function(array_ptr++);
    printf ("%d\n", *array_ptr);
  }
  return 0;
}

void function(int *i) {
  *i = *i + 1;
}
```

   A. 11 21 31 41
   B. 20 30 40 50
   C. 21 31 41 51
   D. 10 20 30 40

5. Write a function named *verify_ numbers* that checks if the second integer is a multiple of the first for a given pair of integers. The function should have two integer formal parameters and return true if the second is a multiple of the first, and false otherwise.

Use this function in a C program that receives three pairs of integers read from the keyboard, as input.

6. Write two functions named *max* and *min* that determine the largest element of an array of integers, and the smallest element of an array of integers, respectively. The functions should have two formal parameters (the size of the array and the array) and return the maximum value and the minimum value of the array's elements. Use these functions in a C program that inputs an array of 10 elements read from the keyboard.

7. Write a function named *perfect_square* that determinates if an integer is a perfect square. The function should have an integer formal parameter and return true if the parameter is a perfect square, and false otherwise. Apply this function to an array of integers, and extract all perfect squares and place them in another array.

8. Write a function that replaces the contents of two double variables with the minimum of the two values. Use the function in a C program that inputs the numbers from the keyboard repeatedly until a special key is pressed. The program should display the values of the variables before and after the function is called.

9. Write a function that calculates the number of the day of a year, and the number of days to the end of that year. Use this function in a C program that inputs a year, a month and a day of the month from the keyboard.

10. Write a text-based, menu-driven C program that allows the user to choose whether to calculate the carbon footprint of a plane, a car and a bicycle. Then, the user inputs a distance in kms and the program displays the appropriate result. Write three functions to calculate the carbon footprint of a plane, a car and a bicycle. Use an array of function pointers to solve the request.

11. Write a C program that calculates the sum of digits of a positive integer number using recursion.

12. Write a C program that finds the reverse of an integer number using recursion.

13. Write a C program that tests up to 10 functions defined in the C Standard Library.

5.4 References

1. Paul Deitel, Harvey Deitel, *C How to Program*, 2010, Sixth edition, Pearson Education, ISBN: 978-0-13-612356-9.
2. C Standard Library header files, https://en.cppreference.com/w/c/header, Accessed in September 2024.

Laboratory paper 6

Dynamic Memory Allocation and Modular Programming

6.1 Overview

- Presentation of the dynamic memory allocation built-in library functions
- Presentation of variables' scope and modular programming
- Use of dynamic memory allocation and modular programming to develop a complex C program
- Work time: 2 hours

6.2 Theoretical Considerations

6.2.1 Dynamic memory allocation

The C programming language handles memory in three ways: statically, automatically, or dynamically [1].

Static variables and data are allocated in a special area of the main memory, named *data segment* and remain in existence for the entire duration of the program.

Automatic variables and data are allocated in the *stack* segment and and they are created and destroyed as functions are called and returned.

When dealing with static and automatic variables and data, the size of the allocated memory must be compile-time constant. If the necessary size of allocated memory is unknown until run-time (i.e., if data of variable size are being read from a user or a file), relying on fixed-size variables and data becomes insufficient. Also, the lifetime of allocated memory can raise concerns. The automatic allocated variables and data cannot survive across multiple function calls, whereas static allocated variables and data exist for the entire duration of the program, regardless of whether they are needed [1]. Dynamic memory allocation mitigates these limitations.

Dynamic memory allocation allows manual memory management using a special area of the main memory, named *heap*. It can lead to memory fragmentation.

C has several built-in functions to perform dynamic memory allocation, listed in the <**stdlib.h**> header file:

- For memory allocation: malloc, calloc, realloc;
- For memory deallocation: free.

These functions are used to allocate a block of memory on the *heap* segment pointed to by the pointer to void returned by these functions. When the block of memory is no longer required, the pointer is freed, which deallocates the block of memory on the heap, making it available for other uses.

malloc, calloc, realloc

The **malloc** function allocates the requested memory and returns a pointer to it.

The prototype of the **malloc** function is:

```
1 void *malloc(size_t size);
```

The **malloc** function has a formal parameter of type size_t representing the size of the allocated memory block, in bytes.

The **malloc** function returns a pointer to void to the allocated block of memory (the address of the first byte in the block) or NULL if the request could not be satisfied. Because the pointer returned is a pointer to void, a type cast is required when storing it into a regular typed pointer.

The **calloc** function allocates the requested memory, initializes it with zero and returns a pointer to it.

The prototype of the **calloc** function is:

```
void *calloc(size_t nitems, size_t size);
```

The **calloc** function has two formal parameters of type size_t representing the number of items of a particular data type to be allocated, and respectively the size of each individual item, in bytes.

The **calloc** function returns a pointer to void to the allocated block of memory (the address of the first byte in the block) or NULL if the request could not be satisfied. Because the pointer returned is a pointer to void, a type cast is required when storing it into a regular typed pointer.

The **realloc** function attempts to resize the block of memory that was previously allocated by a call of the **malloc** or **calloc** functions.

The prototype of the **realloc** function is:

```
void *realloc(void *ptr, size_t size);
```

The **realloc** function has two formal parameters representing a pointer to void to a block of memory previously allocated with **malloc**, **calloc** or **realloc** functions (pointer to void returned by malloc, calloc or realloc calls), and respectively the size of the new allocated block of memory, in bytes.

The **realloc** function returns a pointer to void to the new allocated block of memory (the address of the first byte in the block) or NULL if the request could not be satisfied.

free

The **free** function deallocates the bock of memory previously allocated by a call of the malloc, calloc or realloc functions.

The prototype of the **free** function is:

```
void free(void *ptr);
```

The **free** function has a formal parameter representing a pointer to void to the starting address of the existing allocated memory block (pointer to void returned by malloc, calloc or realloc calls).

The **free** function does no return any value.

Listing 6.1 presents the use of **malloc**, **realloc** and **free** functions to allocate/deallocate a block of memory for an array of integers with a number of elements inserted by the user at run-time.

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

int main() {
int number   = 0;
  int i          = 0;
  int sum        = 0;
  int new_size = 0;
  int *ptr       = NULL;

  printf("Enter the number of elements of an array: ");
  scanf("%d", &number);
  ptr = (int*)malloc(number * sizeof(int));  /* a block of memory is
                                                allocated on the heap */

```

```c
  if(ptr == NULL) {
    printf("Error! Memory not allocated.");
    exit(0);
  }

  printf("Enter the elements of the array: ");
  for(i = 0; i < number; i++) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }
  printf("Sum of the array's elements is: %d.", sum);

  printf("\nThe addresses of the allocated memory are:\n");
  for(i = 0; i < number; i++)
    printf("Address of the %d element with value %d is %p.\n", i, *(ptr + i)
    , ptr + i);

  printf("\nEnter the new size of the array: ");
  scanf("%d", &new_size);
  ptr = realloc(ptr, new_size);  /* the previous allocated block of memory
                                    is resized */
  if(ptr == NULL) {
    printf("Error! Memory not allocated.");
    exit(0);
  }

  sum = 0;
  printf("Enter the new elements of the array: ");
  for(i = 0; i < new_size; i++) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }
  printf("Sum of the array's elements is: %d.", sum);

  printf("\nThe addresses of the new allocated memory are: \n");
  for(i = 0; i < new_size; i++)
    printf("Address of the %d element with value %d is %p.\n", i, *(ptr + i)
    , ptr + i);

  free(ptr);   /* the previous allocated block of memory is deallocated */
  return 0;
}
```

Listing 6.1: Program L6Ex1.c

6.2.2 Variables' scope

Global variables

Global variables are defined at the beginning of a source file (i.e., before any function definition). These variables are accessible from the point of definition to the end of the source file. In a program that utilizes multiple source files, a global variable defined in one source file can be used in all the other source files by declaring it with the *extern* keyword before the variable's data type.

Global variables are allocated at compile-time on the *data segment*.

A *static global variable* is visible in the whole source file, but it cannot be declared extern (and thus made visible) in other files. Its definition uses the *static* keyword before the data type of the variable.

Local variables

Local variables are defined in a function or in a block of code and are visible only within the segment where they were defined. The local variables can be of following types:

- Static variables – allocated at compile-time on the *data segment*. These variables remain in memory while the program is running. Their definition uses the *static* keyword before the data type of the variable;
- Automatic variables – allocated at run-time on the *stack*. These variables are discarded upon return from a function, or at the end of the block of code.

6.2.3 Modular programming

Modular programming is a method to structure large programs in smaller parts, namely *modules* [2]. A module contains related functions, as they are developed for solving a subproblem of a complex problem.

Every module has a well defined *interface* (header file - .h) and an *implementation* part (source file - .c). The module interface specifies how services (functions) provided by the module are made available to client programs. The module implementation hides the code and any other private implementation details from client programs that should not have access to them.

Module interface

Every module interface file (header file - .h) should start with simply C comments that present its purpose, author information, copyright statement, version number and how to check for further updates. The header file contains only definitions of constants, user-defined types, declarations of global variables, and function prototypes that client programs are allowed to access and use.

The definitions/declarations must be enclosed between preprocessor directives, foreseeing in this way the same definitions/declarations from being parsed twice in the same compilation run.

Module implementation

Every module implementation file (source file - .c) should include the required headers and then its own header file. Including its own header file, the code file allocates and initializes the global variables declared in the header. Another useful effect of including the header is that function prototypes are checked against the actual functions, so that for example if some argument in the prototype is forgotten, or if the code is changed and no update of the header is performed, then the compiler will detect the mismatch and inform the programmer with a proper error message.

The constants and the user-defined types defined inside a source file cannot be used by client programs, they are private. The global variables and functions for internal use are defined using the *static* keyword to make them private.

Main program

The main program is a source file that does not require a header file, and contains the only the *main()* function, that does not have a prototype. This source file includes and initializes all the required modules, and finally terminates them once the program is finished.

Listings 6.2, 6.3, 6.4 present a modularized C program example that computes the product of two matrices using pointers and dynamic memory allocation.

```c
/*************************************************************/
/* matrix.h - Header file for module matrix                  */
/* Copyright: 2024                                           */
/* Author:    -                                              */
/* Version:   09-11-2024                                     */
/* Updates:   -                                              */
/*************************************************************/
#ifndef MATRIX_H_INCLUDED
#define MATRIX_H_INCLUDED

/* Headers required by the following definitions/declarations: */
#include <stdlib.h>

/* Constants definitions: */

/* User-defined types definitions: */

/* Global variables declarations: */

/* Function prototypes: */
extern int  **alloc_matrix(int, int);
extern void free_matrix(int**, int);
extern int  read_matrix(int**, int, int, const char*);
extern void print_matrix(int**, int, int, const char*);
extern int  **multiply_matrix(int**, int**, int, int, int);

#endif // MATRIX_H_INCLUDED
```

Listing 6.2: matrix.h

```c
/*****************************************************/
/* matrix.c -  See matrix.h for copyright and info   */
/*****************************************************/
/* System headers and application specific headers:  */
#include "matrix.h"

/* Private constants definitions: */

/* Private user-defined types definitions: */

/* Private global variables definitions: */

/* Public global variables definitions: */

/* Implementation of the private functions: */

/* Implementation of the public functions: */
/*************************************************************/
/* FUNCTION NAME: alloc_matrix                               */
/* DESCRIPTION:   allocates memory for a m by n matrix of integers */
/* ARGUMENT LIST:                                            */
/* Argument      Type     IO  Description                    */
/* ------------ -------- --- ---------------------------------- */
/* m             int      I   number of rows for matrix      */
/* n             int      I   number of columns for matrix   */
/* RETURN VALUE: int **   pointer to allocated area or NULL on failure */
/* CHANGES: -                                                */
/*************************************************************/
```

```c
int **alloc_matrix(int m, int n) {
  int **matrix = NULL;
  int i = 0;

  matrix = (int **) malloc(m * sizeof(int *));
  if (!matrix) return NULL;
  for (i = 0; i < m; i++) {
    *(matrix + i) = (int *) malloc(n * sizeof(int));
    if (!(*(matrix+i))) return NULL;
  }
  return matrix;
}

/*****************************************************************/
/* FUNCTION NAME: free_matrix                                    */
/* DESCRIPTION:   deallocates memory for a m by n matrix of integers */
/* ARGUMENT LIST:                                                */
/* Argument      Type     IO  Description                        */
/* ------------ -------- --  ---------------------------        */
/* matrix        int **   I   matrix to free                     */
/* m             int      I   number of rows for matrix          */
/* RETURN VALUE: void                                            */
/* CHANGES: memory area freed can no longer be used              */
/*****************************************************************/
void free_matrix(int **matrix, int m) {
  int i = 0;

  for (i = 0; i < m; i++) {
    free(*(matrix + i));
  }
  free(matrix);
}

/*****************************************************************/
/* FUNCTION NAME: read_matrix                                    */
/* DESCRIPTION:   reads a matrix of integers                     */
/* ARGUMENT LIST:                                                */
/* Argument      Type         IO   Description                   */
/* ------------ ----------  --   ------------------------------ */
/* matrix        int **       I    pointer to memory area where  */
/*                                 to store read data            */
/* m             int          I    number of rows for matrix     */
/* n             int          I    number of columns for matrix  */
/* name          const char*  I    name to be displayed for each */
/*                                 element of the matrix         */
/* RETURN VALUE: int  1 on success, 0 otherwise                  */
/* CHANGES: memory area previously reserved for storing          */
/*          the read matrix                                      */
/*****************************************************************/
int read_matrix(int **matrix, int m, int n, const char *name) {
  int i = 0, j = 0;

  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++) {
      printf("%s[%d][%d] = ", name, i, j);
      if (1 != scanf("%d", *(matrix + i) + j))
        return 0;
    }
  return 1;
}

```

```c
/*****************************************************************/
/* FUNCTION NAME: print_matrix                                   */
/* DESCRIPTION:   prints a matrix preceeded by a name given to it */
/* ARGUMENT LIST:                                                */
/* Argument      Type         IO  Description                    */
/* ------------ ---------     --  ------------------------------ */
/* matrix        int **       I   matrix to print                */
/* m             int          I   number of rows for matrix to print */
/* n             int          I   number of columns for matrix to print */
/* name          const char*  I   name of matrix                 */
/* RETURN VALUE: void                                            */
/* CHANGES: -                                                    */
/*****************************************************************/
void print_matrix(int **matrix, int m, int n, const char *name) {
  int i = 0, j = 0;

  printf("Matrix %s is:\n-------------------------\n", name);
  for (i = 0; i < m; i++) {
    for (j = 0; j < n; j++) {
      printf("%d ", *(*(matrix + i) + j) );
    }
    printf("\n");
  }
  printf("---------------------------------\n");
}

/*****************************************************************/
/* FUNCTION NAME: multiply_matrix                                */
/* DESCRIPTION:   multiplies matrices matrix1 and matrix2        */
/* ARGUMENT LIST:                                                */
/* Argument      Type      IO  Description                       */
/* ------------ -------    --  ------------------------------    */
/* matrix1       int **     I   matrix matrix1                   */
/* matrix2       int **     I   matrix matrix2                   */
/* m             int        I   number of rows for matrix1       */
/* n             int        I   number of columns for matrix1    */
/*                             and of rows for matrix2           */
/* p             int        I   number of columns for matrix2    */
/* RETURN VALUE: int **   a newly allocated matrix holding      */
/*                        the product of matrix1 and matrix2     */
/* CHANGES: -                                                    */
/*****************************************************************/
int **multiply_matrix(int **matrix1, int **matrix2, int m, int n, int p) {
  int i = 0, j = 0, k = 0;
  int **result = alloc_matrix(m, p); //alocate result matrix

  for (i = 0; i < m; i++)
    for (j = 0; j < p; j++)
      *(*(result + i) + j) = 0;
  for (i = 0; i < m; i++)
    for (j = 0; j < n; j++)
      for (k = 0; k < p; k++)
        *(*(result + i) + k) += (*(*(matrix1 + i) + j)) * (*(*(matrix2 + j)
   + k));
  return result;
}
```

Listing 6.3: matrix.c

```c
/*************************************************************/
/* Complex application                                       */
/* Copyright: 2024                                           */
/* Author:    -                                              */
/* Version:   09-11-2024                                     */
/* Updates:   -                                              */
/*************************************************************/
/* Include standard headers: */
#include <stdio.h>
#include <stdlib.h>

/* Include modules header that are directly invoked: */
#include "matrix.h"

int main() {
  int m = 0, n = 0, p = 0;              //matrices dimensions
  int **matrix1, **matrix2, **result;   //matrices

  /* Read the dimensions of the matrices */
  printf("Please input the number of rows of the First matrix: ");
  scanf("%d", &m);
  printf("Please input the number of columns/rows of the First matrix/Second
     matrix: ");
  scanf("%d", &n);
  printf("Please input the number of columns of the Second matrix: ");
  scanf("%d", &p);

  /* Allocate memory for matrices */
  matrix1 = alloc_matrix(m, n);
  matrix2 = alloc_matrix(n, p);

  /* Read and print matrices matrix1 and matrix2 */
  read_matrix(matrix1, m, n, "First matrix");
  print_matrix(matrix1, m, n, "First matrix");
  read_matrix(matrix2, n, p, "Second matrix");
  print_matrix(matrix2, n, p, "Second matrix");

  /* Multiply matrix1 by matrix2 and print the result */
  result = multiply_matrix(matrix1, matrix2, m, n, p);
  print_matrix(result, m, p, "Product");

  /* Deallocate memory */
  free_matrix(result, m);
  free_matrix(matrix2, n);
  free_matrix(matrix1, m);
  return 0;
}
```

Listing 6.4: main.c

Code documentation tools

When dealing with large programs a code documentation generator tool should be used to provide a comprehensive description of the code (i.e., its purpose, functionality, usage). This documentation helps the developers to understand the architecture and logic behind the code, which leads to easier maintenance and cooperation.

Doxygen [3] is a widely-used documentation generator tool that can be used for C programs. It generates automatically the documentation in formats like HTML, PDF, Word and XML. The documentation consists of source code comments, information about

the functions, and variables. By its cross-referencing capabilities, Doxygen allows easy navigation between different parts of the documentation.

6.3 Lab Tasks

1. Run the program given as example and analyze the output.
2. Write a modular program of your choice.

6.4 References

1. C dynamic memory allocation, https://en.wikipedia.org/wiki/C_dynamic_memory_allocation, Accessed in September 2024.
2. Modular programming in C, https://www.icosaedro.it/c-modules.html, Accessed in September 2024.
3. Doxygen, https://www.doxygen.nl/, Accessed in October 2024.

Laboratory paper 7

Strings in C

7.1 Overview

- Presentation of strings in C
- Use of strings within simple C programs
- Work time: 2 hours

7.2 Theoretical Considerations

A string in C is a one dimensional array of characters that ends with a null character '\0' (NUL). Each character of a string is represented on a byte by its ASCII (American Standard Code for Information Interchange) Code.

7.2.1 String variable definition and initialization

The following syntaxes is used to define a string variable in C:

char identifier[integer_constant];
char identifier[] = {'char_0', 'char_1', ..., 'char_n', '\0'};
char char identifier[] = "char_0char_1...char_n";

where:

- *identifier* is the name of the string;
- *integer_constant* is the size of the string given as an integer constant greater than 0;
- *char_0, char_1, ..., char_n* are the characters of the string.

From the previously presented syntaxes, it appears that the size of a string can be omitted when defining it, but in this case the string must be initialized.

A string can also be defined with the help of pointers. In this case, the name of the array that holds the string is considered a constant pointer to the character string, as the syntax shows:

char *identifier;

where:

- * is the symbol used to define a pointer (in this case * is not the dereference operator);
- *identifier* is the name of the string.

This definition of a string requires initialization. In the following, three ways for doing this are presented.

Initialization with the address of a static string
char identifier1[3] = "cp"; char *identifier = &identifier1;
Initialization with the address of a constant string
char *identifier = "char_0char_1...char_n";
Initialization with the address of a dynamic allocated memory block
char *identifier = (char *)malloc(100 * sizeof(char));

Several definitions of string variables are presented below.

```
1 char string[]  = "computer";
2 char string1[] = {'c', 'o', 'm', 'p', 'u', 't', 'e', 'r', '\0'};
3 char *string2  = "computer";
```

```
4 char string3[10];
5 char *string4  = (char *)malloc(10 * sizeof(char));
```

7.2.2 Internal memory representation of a string

Figure 7.1 presents graphically the way in which a string is representing in the computer memory. Each character is represented in binary using the ASCII Code (i.e. 'c') -> 01100011.

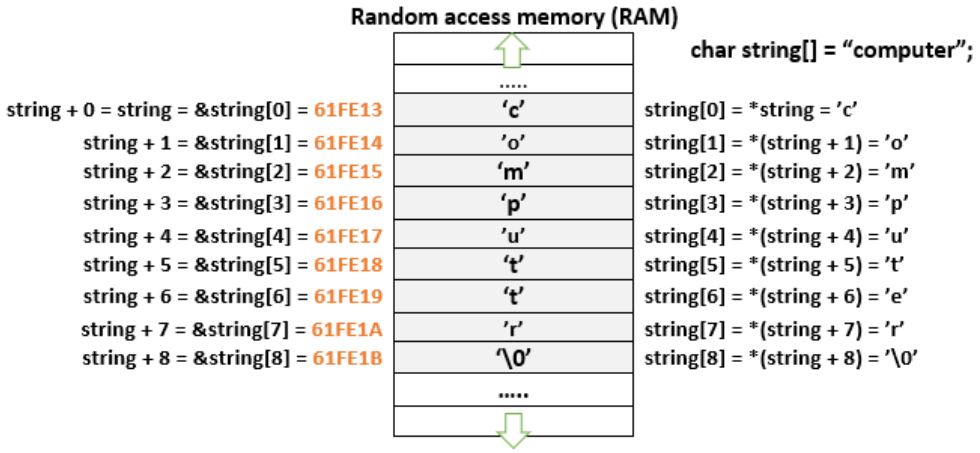


Figure 7.1: Memory representation of a string definition and initialization

The notations (subscript and offset) used in Figure 7.1 can be generalized as follows [1]:

- string[i], where i ∈ [0, size of string] represents the ASCII Code of the i^{th} character of the string;
- string + i, where i ∈ [0, size of string] is the address of the i^{th} character of the string;
- *(string + i) has the same effect as string[i].

Listing 7.1 displays each character within the string and their addresses using the subscript and offset notations.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5     int  i        = 0;
6     char string[] = {"computer"};
7
8     for(i = 0; i < 9; i++)
9         printf("string[%d] = %c  address = %p\n", i, string[i], string + i);
10    return 0;
11 }
```

Listing 7.1: Program L7Ex1.c

7.2.3 Array of strings

Multiple strings can be grouped to form an array of strings. An array of strings is a two dimensional array that may be defined in C as:

char identifier[NUMBER_OF_STRINGS][MAX_STRING_SIZE] = {"string_0", "string_1", ..., "string_number_of_strings - 1"};

where:

- *identifier* is the name of the array of strings;
- *NUMBER_OF_STRINGS* is the number of the strings that form the array;
- *MAX_STRING_SIZE* is the size of the longest string in the array;
- *string_0, string_1, ..., string_number_of_strings - 1* are the strings of the array.

Another way to define an array of strings is by using an array of pointers.

char *identifier[]={"string_0", "string_1", ..., "string_n"};

where:

- * is the symbol used to define a pointer (in this case * is not the dereference operator);
- *identifier* is the name of the array of strings (identifier[i], for i ∈ [0, n] is a pointer to the *string_i*);
- *string_0, string_1, ..., string_n* are the strings of the array.

Several definitions of arrays of strings are presented below.

```
1 char array_strings1[2][9] = {{'t', 'e', 's', 't', '\0'},
2                              {'c', 'o', 'm', 'p', 'u', 't', 'e', 'r', '\0'}};
3 char array_strings2[2][9] = {"test", "computer"};
4 char *array_strings[]     = {"test", "computer"};
```

7.2.4 Internal memory representation of an array of strings

Figure 7.2 presents graphically the way in which an array of strings, defined using an array of pointers, is represented in the computer memory. The notations (subscript and offset) used in Figure 7.2 can be generalized as follows:

- array_strings + i points to the i^{th} string or the array;
- *(array_strings + i) + j points to the j^{th} character of the i^{th} string (*(array_strings + i) + j is a pointer to char (char *));
- *(*(array_strings + i) + j) gets the element at the j^{th} character of the i^{th} string and has the same effect as array_strings[i][j].

7.2.5 Standard string processing functions

The C programming language has a set of built-in functions that deal with string processing. These are included in the <**string.h**> header file. The most commonly used are the following:

- For string length: strlen;
- For string copy: strcpy, strncpy;
- For string concatenation: strcat, strncat;
- For string comparison: strcmp, strncmp;
- For string search: strchr, strstr.

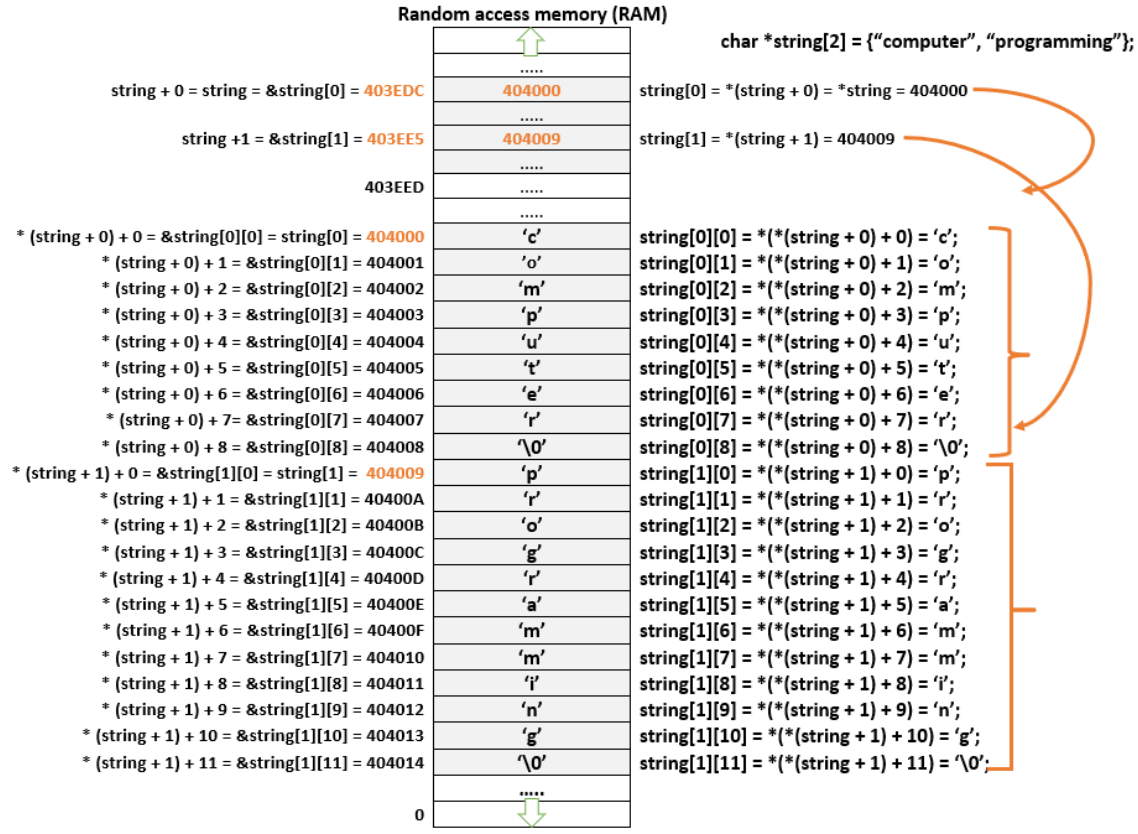


Figure 7.2: Memory representation of an array of two arrays of characters (strings) definition and initialization

strlen

The **strlen** function computes the length (size) of a string up to but not including the terminating '\0' character.

The prototype of the **strlen** function is:

```
size_t strlen(const char *str);
```

The **strlen** function has a formal parameter *str* of type const char * representing the string, and returns an unsigned integer value of type size_t representing the length of the string.

strcpy and strncpy

The **strcpy** function copies a source string to a destination string. The copy includes the '\0' character.

The prototype of the **strcpy** function is:

```
char *strcpy(char *dest, const char *src);
```

The **strcpy** function has two formal parameters, *src* of type const char * representing the source string, respectively *dest* of type char * representing the destination string, and returns the address of the destination string.

The **strncpy** function copies a specified number of characters from a source string to a destination string. After the last character that is transferred, the '\0' character must be

appended. If the number of characters, which is to be copied, is greater than the length of the source string, the entire source string is copied.

The prototype of the **strncpy** function is:

```
char *strncpy(char *dest, const char *src, size_t n);
```

The **strncpy** function has three formal parameters, *src* of type const char * representing the source string, *dest* of type char * representing the destination string, respectively *n* of type size_t representing the number of characters to be copied, and returns the address of the destination string.

strcat and strncat

The **strcat** function appends a source string to the end of a destination string. After the last character that is appended, the '\0' character must be inserted.

The prototype of the **strcat** function is:

```
char *strcat(char *dest, const char *src);
```

The **strcat** function has two formal parameters, *src* of type const char * representing the source string, respectively *dest* of type char * representing the destination string, and returns the address of the destination string.

The **strncat** function appends at most a specified number of characters of a source string to the end of a destination string. The '\0' character is automatically appended to the result string. If the number of characters, which is to be appended, is greater than the length of the source string, the function has the effect as the **strcat** function.

The prototype of the **strncat** function is:

```
char *strncat(char *dest, const char *src, size_t n);
```

The **strncat** function has three formal parameters, *src* of type const char * representing the source string, *dest* of type char * representing the destination string, respectively *n* of type size_t representing the number of characters to be appended, and returns the address of the destination string.

strcmp and strncmp

The **strcmp** function compares two strings.

The prototype of the **strcmp** function is:

```
int strcmp(const char *str1, const char *str2);
```

The **strcmp** function has two formal parameters, *str1* and *str2* of type const char * representing the strings which are compared, and returns a negative value if the string having the address *str1* is less than the string having the address *str2*, 0 if the two strings are equal, and a positive value if the string having the address *str1* is greater than the string having the address *str2*.

The **strncmp** function compares a specified number of characters from two strings.

The prototype of the **strcmp** function is:

```
int strncmp(const char *str1, const char *str2, size_t n);
```

The **strncmp** function has three formal parameters, *str1* and *str2* of type const char * representing the strings from where *n* number of characters of type size_t are compared, and returns a negative value if the string having the address *str1* is less than the string having the address *str2*, 0 if the two strings are equal or if their first *n* characters are equal,

and a positive value if the string having the address *str1* is greater than the string having the address *str2*.

There is a frequent error that occurs when dealing with the comparison of two strings. The next listings highlight the difference between the use of the simple assignment operator and of the **strcmp** function.

```
char a[50] = {'\0'};
char b[50] = {'\0'};
scanf("%s%s", a, b);
if(a == b) printf("The strings are equal.\n"); //Wrong - the addresses of a
    and b are compared
```

```
char a[50] = {'\0'};
char b[50] = {'\0'};
scanf("%s%s", a, b);
if(strcmp(a, b) == 0) printf("The strings are equal.\n"); //Correct - the
    values of a and b are compared
```

strchr and strstr

The **strchr** function searches for the first occurrence of a character in a string.

The prototype of the **strchr** function is:

```
char *strchr(const char *str, int c);
```

The **strchr** function has two formal parameters, *str* of type const char * representing the string and *c* of type integer representing the character to be searched, and returns the address of the first occurrence of character in string.

The **strstr** function searches for the first occurrence of a string in another string.

The prototype of the **strchr** function is:

```
char *strstr(const char *str, const char *substr);
```

The **strstr** function has two formal parameters, *str* of type const char * representing the source string, *substr* of type const char * representing the string to be searched in the source string, and returns the address of the first occurrence of string *substr* in string *str*.

Listing 7.2 presents a C program which exemplifies the use of some string processing built-in functions.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_LENGTH 50

int main() {
  char src_string[]             = {"computer programming"};
  char dest_string[MAX_LENGTH]  = {'\0'};
  char sub_string[MAX_LENGTH]   = {'\0'};
  char *string                  = NULL;
  char ch                       = '\0';
  int  nch                      = 0;

  printf("The length of the source string is: %u\n", strlen(src_string));
  printf("----------------------------------------\n");
  printf("The source string: %s\n", src_string);
  printf("The destination string before copying the source string: %s\n",
    dest_string);
  strcpy(dest_string, src_string);
  printf("The destination string after copying the source string: %s\n",
    dest_string);
```

```
20   printf("-----------------------------------------\n");
21   if(strcmp(dest_string, src_string) == 0)
22     printf("Source string and destination string are equal.\n");
23   printf("-----------------------------------------\n");
24   printf("Insert how many characters to copy from the source string to
       substring: ");
25   scanf("%d", &nch);
26   strncpy(sub_string, src_string, nch);
27   printf("The substring after copying %d characters from the source string:
      %s\n", nch, sub_string);
28   printf("Insert how many characters to append from the source string to
       destination string: ");
29   scanf("%d", &nch);
30   strncat(dest_string, src_string, nch);
31   printf("The destination string after appending %d characters from the
       source string: %s\n", nch, dest_string);
32   printf("Insert how many characters to be compared from destination string
       and substring: ");
33   scanf("%d", &nch);
34   if(strncmp(dest_string, sub_string, nch) == 0)
35     printf("The first %d characters from the destination string and the
       substring are equal.\n", nch);
36   else
37     printf("The first %d characters from the destination string and the
       substring are not equal.\n", nch);
38   printf("-----------------------------------------\n");
39   printf("Insert a character to be searched in substring: ");
40   scanf(" %c", &ch);
41   string = strchr(sub_string, ch);
42   printf("Substring after the first %c character is: %s", ch, string + 1);
43   return 0;
44 }
```

Listing 7.2: Program L7Ex2.c

7.3 Lab Tasks

1. Run the programs given as examples and analyze the output.
2. Write a function that removes a substring from an initial string by indicating the starting position and the length of the substring to be deleted. Use the function in a C program that inputs from the keyboard the starting position and the length of the substring to be deleted from the keyboard.
3. Write a C program to read several strings form the keyboard and display the lexicographically largest and smallest strings. Use strcpy and strcmp built-in functions.
4. Write a C program to read several strings from the keyboard and display the longest string. Use dynamic memory allocation.

7.4 References

1. Iosif Ignat, *Programarea calculatoarelor: indrumator de lucrari de laborator*, 2003, Second edition, U.T.Press, Cluj-Napoca, ISBN: 973-662-024-7.

Laboratory paper 8

Structures, Unions and Enumerations in C

8.1 Overview

- Presentation of user-defined data types: structure, union, and enumeration
- Use of user-defined data types to develop C programs
- Work time: 2 hours

8.2 Theoretical Considerations

Structures, unions, and enumerations are C programming language elements that allow the definition of new data types.

8.2.1 Structures

Structures provide a method for storing various values in variables that can have different data types, all under a single name.

Definition

The following syntax is used to define a structure in C:

```
struct structure_identifier {
   data_type structure_member1_identifier;
   data_type structure_member2_identifier;
   ......
   data_type structure_membern_identifier;
};
```

where:

- *structure_identifier* is the name of the structure;
- One or more members, separated by semicolon and specified by:
  - *structure_member_identifier*.

Several structure definitions are presented below:

```
1  struct book {
2    int  book_id;
3    char title[50];
4    char author[50];
5  };
6
7  struct address {
8    char *city;
9    char *country;
10 };
11
12 struct student {
13   int     student_id;
14   int     student_age;
15   char    *student_name;
16   struct address student_address;
17 };
```

Definition and initialization of structure variables

Storage is allocated for a structure when a variable of that structure is defined.

A structure variable can either be defined through a structure definition, or as a separate definition like basic data types.

The following syntaxes are used to define structure variables in C:

struct structure_identifier {
  data_type structure_member1_identifier;
  data_type structure_member2_identifier;
  .....
  data_type structure_member$n$_identifier;
} variable1_identifier [, variable2_identifier, ... , variable$n$_identifier];

struct structure_identifier variable1_identifier [, variable2_identifier, ... , variable$n$_identifier];

Several structure variables definitions are presented below:

```
struct book {
  int  book_id;
  char title[50];
  char author[50];
} book1, book2;

struct book book3, book4;
```

Structure variables can be initialized using curly braces. Each member of the structure receives a proper value. If not all members of a structure variable have been initialized, the uninitialized ones will be automatically set to 0, NUL if the member is a character, or NULL if the member is a pointer. The variables of a structure defined outside a function definition are automatically set to 0, NUL or NULL if they are not explicitly initialized.

The variables of a structure may also be initialized through assignment statements. In this case, a structure variable of the same type can be assigned, or values can be assigned to the individual members of the structure.

Two examples of structure variables initializations are presented below:

```
struct book {
  int  book_id;
  char title[50];
  char author[50];
};

struct book book1 = {1, "C How to Program", "Deitel"};
struct book book2 = book1;
```

Accessing structure members

The following two C operators are used to access the members of a structure [1]:

- The structure member operator (.), also called the dot operator. This operator accesses a structure member using a structure variable name.
- The structure pointer operator (->), also called the arrow operator. This operator accesses a structure member using a pointer to the structure.

Listing 8.1 presents a C program where the members of a structure are accessed using both operators.

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

struct book {     /* define data type struct book */
  int  book_id;
  char title[50];
  char author[50];
};

void print_struct(struct book);

int main() {
  struct book book1 = {1, "C How to Program", "Deitel"};   /* define and
                          initialize book1 a variable of type struct book */
  print_struct(book1);
  book1.book_id = 2;
  strcpy(book1.title, "C Programming");
  strcpy(book1.author, "Smith");
  print_struct(book1);
  return 0;
}

void print_struct(struct book bk) {
  struct book *bk_ptr = &bk;
  printf("%d; %s; %s\n", bk.book_id, bk.title, bk.author);
  printf("%d; %s; %s\n", bk_ptr->book_id, bk_ptr->title, bk_ptr->author);
  printf("%d; %s; %s\n", (*bk_ptr).book_id, (*bk_ptr).title, (*bk_ptr).
    author);
}
```

Listing 8.1: Program L8Ex1.c

Listing 8.2 presents a C program that dynamically allocates memory for a varying number of a structure variables inserted by the user from the keyboard (this forms an array of structure variables).

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

struct stock {
  char  code[3];
  char  name[30];
  int   quantity;
  float price;
};

void insert_products(int, struct stock*);
void display_products(int, struct stock*);

int main() {
  struct stock *ptr;
  int    number = 0;

  printf("Enter the number of products: ");
  scanf("%d", &number);

  ptr = (struct stock*) malloc(number * sizeof(struct stock));
  insert_products(number, ptr);

  printf("Displaying the products with quantity > 0:\n");
  display_products(number, ptr);
```

```
26
27    free(ptr);
28    return 0;
29  }
30
31  void insert_products(int number, struct stock *st){
32    int i = 0;
33    for(i = 0; i < number; i++) {
34      printf("Enter code, name, quantity, price of the %d product:\n", i + 1);
35      scanf("%s%s%d%f", (st + i) -> code, (st + i) -> name, &(st + i) ->
        quantity, &(st + i) -> price);
36    }
37  }
38
39  void display_products(int number, struct stock *st){
40    int i = 0;
41    for(i = 0; i < number; i++) {
42      if ((st + i) -> quantity > 0)
43        printf("%s\t%s\t%d\t%.2f\n", (st + i) -> code, (st + i) -> name, (st +
        i) -> quantity, (st + i) -> price);
44    }
45  }
```

Listing 8.2: Program L8Ex2.c

8.2.2 Unions

At different times during execution, the same memory block can hold different data types. This is useful for saving memory. This can be achieved by grouping all data which will be allocated in the same memory block. The data types obtained in this way are called *unions*. In practice, the unions are rarely used in practice.

Definition

The following syntax is used to define a union in C:

```
union union_identifier {
  data_type union_member1_identifier;
  data_type union_member2_identifier;
  ......
  data_type union_membern_identifier;
};
```

where:

- *union_identifier* is the name of the union;
- One or more members, separated by semicolon and specified by:
  - *union_member_identifier*.

Two union definitions are presented below:

```
union test1 {
  int  x;
  char y;
};
```

```
union test2 {
  int  arr[10];
  char y;
};
```

At any given time only one member of a union can contain a value, although the union in question has many members. The size of a union is taken according to the size of the largest data type member of the union.

Definition of union variables

A union variable can either be defined through a union definition, or as a separate definition like basic data types.

The following syntaxes are used to define union variables in C:

union union_identifier {
  data_type union_member1_identifier;
  data_type union_member2_identifier;
  .....
  data_type union_member$n$_identifier;
} variable1_identifier [, variable2_identifier, ... , variable$n$_identifier];

union union_identifier variable1_identifier [, variable2_identifier, ... , variable$n$_identifier];

Several union variables definitions are presented below:

```
union test {
  int  x;
  char y;
} test1, test2;

union test test3, test4;
```

Accessing union members

The following two C operators are used to access members of a union [1]:

- The union member operator (.), also called the dot operator. This operator accesses a union member using a union variable name.
- The union pointer operator (->), also called the arrow operator. This operator accesses a union member using a pointer to the union.

Listing 8.3 presents a C program where the members of a union take values and then are accessed using the member operator.

```
#include <stdio.h>

union number {  /* define data type union number */
  int    a;
  double b;
} value;        /* define value as a variable of type union number */

int main(void) {
  value.a = 100;
  printf("You inserted a value in the integer member of the union number and
     print the values of both its members: %d\t%.2f\n", value.a, value.b);
```

```
11   value.b = 100.00;
12   printf("You inserted a value in the double member of the union number and
       print the values of both its members: %d\t%.2f\n", value.a, value.b);
13   return 0;
14 }
```

Listing 8.3: Program L8Ex3.c

8.2.3 Enumerations

Enumerations enable the programmer to use suggestive names for integer constants. The enumeration integer constants behave like symbolic constants with values automatically set. By default, the values of integer constants within an enumeration begin at 0 and are incremented by 1. Also, the values of the integer constants can be explicitly assigned by the programmer using the assignment operator.

Definition

The following syntax is used to define an enumeration in C:

enum enum_identifier {constant1_identifier, ..., constant$n$_identifier};

where:

- *enum_identifier* is the name of the enumeration;
- One or more constants, separated by comma and specified by:
  - *constant_identifier*.

Several enumeration definitions are presented below:

```
1 //R = 0, G = 1, B = 2
2 enum colors {R, G, B};
3
4 //R = 255, G = 255, B = 255
5 enum cls {R = 255, G = 255, B = 255};
6
7 //R = 5, G = 6, B = 10
8 enum cl {R = 5, G, B = 10};
```

Definition of enumeration variables

An enumeration variable can either be defined through an enumeration definition, or as a separate definition like basic data types.

The following syntaxes are used to define enumeration variables in C:

enum enum_identifier {constant1_identifier, ..., constant$n$_identifier}
variable1_identifier [, variable2_identifier, ... , variable$n$_identifier];

enum enum_identifier variable1_identifier [, variable2_identifier, ... , variable$n$_identifier];

Two enumeration variables definitions are presented below:

```
1 enum colors {R, G, B} color;
2 enum colors yellow;
```

Listing 8.4 presents a C program where an enumeration is used.

```c
#include <stdio.h>

int main() {
  enum number {one = 1, two, three, four, five};
  enum number x, y;
  int         z = 0, w = 0;

  x = two;
  y = three;
  z = 2 * x + y;
  w = y - x;
  printf("z = %d w = %d\n", z, w);
  return 0;
}
```

Listing 8.4: Program L8Ex4.c

8.2.4 Defining data types using symbolic names

In the C programming language a symbolic name can be assigned to a predefined data type, or to a user-defined data type, using the *typedef* keyword.

The following syntax is used to assign a symbolic name to a user-defined data type in C:

typedef type {...} type_name;

where:

- *type* could be any derived data type (struct, union or enum);
- *type_name* is the symbolic name that can be assigned to a predefined type.

Several new data type definitions are presented below:

```c
//definition of the Book struct data type
typedef struct {
   char   product_name[50];
   char   product_description[100];
   double product_price;
   int    product_id;
} Product;
//definition of two Product variables
Product product1, product2;

//definition of the Test union data type
typedef union {
  char   x[10];
  double code;
} Test;
//definition of two Test variables
Test variable1, variable2;

//definition of the Boolean enum data type
typedef enum {false, true} Boolean;
//definition of two Boolean variables
Boolean variable1, variable2;
```

8.3 Lab Tasks

1. Run the programs given as examples and analyze the output.

2. Find the error in each of the following sequences of code:
   Assume:

```
struct test{
  char test_name[30];
  char test_description[100];
} a, *aptr;
aptr = &a;
```

   a) printf("%s", *aptr->test_name);
   Assume:

```
struct test {
  char test_name[30];
  char test_description[100];
} b[12];
```

   b) printf("%s", b.test_name); (to print the member test_name of array element 5)
   c)

```
union new {
  int   a;
  float b;
  char  c;
};
union new x = {1.3};
```

   d)

```
struct product {
  int   product_id;
  float product_price;
  char  product_name[50];
};
product x;
```

   Assume the two previous definitions of struct test and struct product:

   e)

```
struct test    test1;
struct product product1;
test1 = product1;
```

3. Write code for each of the following requirements:
   a) Define a structure named *test* with two members: an integer *test_ number* and an array of characters *test_ name* with a size up to 30 characters long.
   b) Define the synonym *Test* for the user-defined data type *struct test*.
   c) Use *Test* to define variables *t* and *t_ array[10]* of type *struct test*, and variable *ptr* of type pointer to *struct test*.
   d) Read from the keyboard values for the individual members of variable *t*.
   e) Assign the value of variable *t* to element 4 of array *t_ array*.
   f) Assign the address of array *t_ array* to the pointer variable *ptr*.
   g) Display the values of element 4 of array *t_ array* using the variable *ptr* and the structure pointer operator.

4. Write a C program that prompts the users for their first and last names and, prints the total number of letters in those names. Use pointer to structure and define the following functions: get_info(), make_info(), show_info().

8.4 References

1. Paul Deitel, Harvey Deitel, *C How to Program*, 2022, Ninth edition, Pearson Education, ISBN: 978-0-13-739839-3.

Laboratory paper 9

Files in C

9.1 Overview

- Presentation of the high level functions used for file management
- Use of high level functions for file management to develop C programs
- Work time: 2 hours

9.2 Theoretical Considerations

Files represent sequences of bytes used to store relevant information. The C programming language can manage two types of files: *text files* and *binary files*.

Text files are standard .txt files that can be easily generated using basic text editors. Their contents can be viewed as plain text and can easily be edited or deleted. These files require minimum effort to maintain, are easily readable, and provide the lowest level of security while occupying more storage space compared to binary files.

Binary files are typically .bin files that store the data in binary form (0's and 1's). These files store higher amounts of data, are not easily readable and provide a better security compared to text files.

The C programming language has several built-in library functions to perform file management at a high level.

A pointer to a structure of type FILE is attached to each file in C:

```
FILE *pointer_identifier;
```

The type FILE and the prototypes of the file management functions are listed in the <**stdio.h**> header file. The most commonly used are the following:

- For creating a new file or for opening an existing one: fopen;
- For closing a file: fclose;
- For writing chars, strings or other data into text files: fputc, fputs, fprintf;
- For reading chars, strings or other data from text files: fgetc, fgets, fscanf;
- For setting the file position indicator of a file to a given offset: fseek;
- For writing items into text and binary files: fwrite;
- For reading items from text and binary files: fread.

9.2.1 fopen

The **fopen** function creates a new file (text or binary) or opens an existing one.

The prototype of the **fopen** function is:

```
1 FILE *fopen(const char *filename, const char *mode);
```

The **fopen** function returns a pointer of type FILE if successful, otherwise it returns NULL.

filename is a string representing the name of the file or the path of the file.

mode is a string representing the access mode of the file which can have one of the values presented in Table 9.1.

The following access modes are used to handle binary files: "rb", "wb", "ab", "rb+", "r+b", "wb+", "w+b", "ab+", "a+b".

Table 9.1: Access modes for text files [1]

Mode	Description
r	Opens an existing text file for reading purpose. If the file does not exist, NULL is returned.
w	Opens a text file for writing. If the file does not exist, then it is created and the program starts writing the data from the beginning of the file. If the file exists, its content is overwritten.
a	Opens a text file for writing in appending mode. If the file does not exist, then it is created and the program starts writing the data from the beginning of the file. If the file exists, the program appends the data to the end of the file.
r+	Opens a text file for both reading and writing. If the file does not exist, NULL is returned.
w+	Opens a text file for both reading and writing. If the file does not exist, it is created. If the file exists, its contents is overwritten.
a+	Opens a text file for both reading and appending. If the file does not exist, it is created. The reading starts from the beginning of the file, but the writing can only be appended.

9.2.2 fclose

The **fclose** function closes a file (text or binary).
The prototype of the **fclose** function is:

```
1 int fclose(FILE *fp);
```

The **fclose** function returns 0 if it successfully closes the file, or EOF (end-of-file) if there is an error in closing it. EOF is a constant defined in the header file <**stdio.h**>.
fp is a pointer of type FILE representing an opened file (returned by **fopen**).

9.2.3 fputc, fputs, fprintf

The **fputc** function writes a character into a text file.
The prototype of the **fputc** function is:

```
1 int fputc(int ch, FILE *fp);
```

The **fputc** function returns the ASCII code of the same character if successful, or EOF if an error occurs.
ch is the ASCII code of the character to write.
fp is a pointer of type FILE representing an opened file (returned by **fopen**).

The **fputs** function writes a string into a text file.
The prototype of the **fputs** function is:

```
1 int fputs(const char *string, FILE *fp);
```

The **fputs** function returns a non-negative value if successful, or EOF if an error occurs.
string is the string to write.
fp is a pointer of type FILE representing an opened file (returned by **fopen**).

The **fprintf** function writes a formatted output into a text file.
The prototype of the **fprintf** function is:

```
int fprintf(FILE *fp, const char *format [, expression list]);
```

The **fprintf** function returns the total number of items written into the file if successful, or a negative value if an error occurs.

fp is a pointer of type FILE representing an opened file (returned by **fopen**).

format is specified as a string enclosed in double quotes ("). It can optionally include embedded format tags that are substituted with the values provided in the *expression list* and formatted as specified, as in the case of the **printf** function.

expression list contains the items to write.

9.2.4 fgetc, fgets, fscanf

The **fgetc** function reads a character from a text file.
The prototype of the **fgetc** function is:

```
int fgetc(FILE *fp);
```

The **fgetc** function returns the ASCII code of the character read from a file if successful, or EOF if an error occurs.

fp is a pointer of type FILE representing an opened file (returned by **fopen**).

The **fgets** function reads a string from a text file.
The prototype of the **fgets** function is:

```
char *fgets(char *string, int n, FILE *fp);
```

The **fgets** function returns the string read from a file if successful, or NULL if an error occurs.

string is the string where the characters read from the file is stored.

n is the maximum number of characters to be read.

fp is a pointer of type FILE representing an opened file (returned by **fopen**).

The **fscanf** function reads a formatted input from a text file. The prototype of the **fscanf** function is:

```
int fscanf(FILE *fp, const char *format [, &variable1] [, &variable2] ...);
```

The **fscanf** function returns the total number of items read from the file if successful, or a negative value if an error occurs.

fp is a pointer of type FILE representing an opened file (returned by **fopen**).

format is specified as a string enclosed in double quotes ("). It contains embedded format tags that specify the data types of the *variable1, variable2, ...*, as in the case of the **scanf** function.

variable1, variable2, ... contains the variables where the items read from a file are stored.

9.2.5 fseek

The file position indicator of a file tracks the location in the file where the next byte will be read or written. It is represented as an integer that counts the number of bytes from the beginning of the file.

The **fseek** function sets the file position indicator of a file to a given offset.
The prototype of the **fseek** function is:

```
int fseek(FILE *fp, long int offset, int whence);
```

The **fseek** function returns 0 if successful, or a non-zero value if an error occurs.
fp is a pointer of type FILE representing an opened file (returned by **fopen**).
offset is the number of bytes to offset from *whence*.
whence specifies the location where the offset starts and has the values presented in Table 9.2.

Table 9.2: Constants that indicate the location where the offset starts

Whence	Description
SEEK_SET	Starts the offset from the beginning of the file.
SEEK_CUR	Starts the offset from the current location of the cursor in the file.
SEEK_END	Starts the offset from the end of the file.

The current position in a file, given as an offset in bytes from its beginning, is returned by the **ftell** function which has the following prototype:

```
long int ftell(FILE *fp);
```

To set the file position indicator of a file to the beginning of the file, the **rewind** function should be used:

```
void rewind(FILE *fp);
```

Listing 9.1 presents a C program which exemplifies the character and string oriented file processing.

```
#include <stdio.h>

int main() {
  char  ch;
  char string[100];
  char file_name[50] = "file.txt";
  int   i            = 1;
  FILE *fp;

  fp = fopen(file_name, "w");
  printf("Please input a text to store in file \"%s\". Press Ctrl+Z to end.\
    n", file_name);
  while((ch = fgetc(stdin)) != EOF) {
    fputc(ch, fp);
  }
  fclose(fp);

  fp = fopen(file_name, "r+");
  fseek(fp, 0, SEEK_END);
  printf("\nPlease input the strings to append to the created file. Press
    Ctrl+Z to end.\n");
  while(fgets(string, 100, stdin) != NULL) {
    fputs(string, fp);
  }
  fclose(fp);

  printf("\nLines of the files (numbered):\n");
  fp = fopen(file_name, "r");
  while(fgets(string, 100, fp) != NULL) {
```

```
28      printf("%d %s", i, string);
29      i++;
30    }
31    fclose(fp);
32    return 0;
33  }
```

Listing 9.1: Program L9Ex1.c

9.2.6 fwrite

The **fwrite** function writes items into a binary or text file.

The prototype of the **fwrite** function is:

```
size_t fwrite(const void *ptr, size_t size_of_elements, size_t
    number_of_elements, FILE *fp);
```

The **fwrite** function returns the total number of items written into the file if successful. If this number differs from the *number_of_elements* parameter, an error occurs.

ptr is a pointer to the array of items to be written.

size_of_elements is the size in bytes of each item to be written.

number_of_elements is the number of items, each one with a size of *size_of_elements* bytes.

fp is a pointer of type FILE representing an opened file (returned by **fopen**).

9.2.7 fread

The **fread** function reads items from a binary or text file.

The prototype of the **fread** function is:

```
size_t fread(void *ptr, size_t size_of_elements, size_t number_of_elements,
    FILE *fp);
```

The **fread** function returns the total number of items read from the file if successful. If this number differs from the *number_of_elements* parameter, an error occurs.

ptr is a pointer to the array of items to be read.

size_of_elements is the size in bytes of each item to be read.

number_of_elements is the number of items, each one with a size of *size_of_elements* bytes.

fp is a pointer of type FILE representing an opened file (returned by **fopen**).

Listing 9.2 presents a C program which exemplifies the binary processing of files.

```
#include <stdio.h>

typedef struct {
  char name[40];
  int  salary;
} Record;

void create_file(const int, const char*);
void show_file(const char*);

int main() {
  char file_name[40] = "file.bin";
  int  n             = 0;

  printf("Input the number of persons, n: ");
  scanf("%d", &n);
```

```c
  create_file(n, file_name);
  printf("\nFile content:\n");
  show_file(file_name);
  return 0;
}

void create_file(const int n, const char *filename) {
  FILE  *fp;
  Record rec;
  int    i = 0;

  fp = fopen(filename, "wb");
  if(fp == NULL) {
    printf("Error!");
    exit(1);
  }

  for(i = 1; i <= n; i++) {
    fflush(stdin); //fflush - clears or flushes the stdin buffer
    printf("First and last name of the person: ");
    fgets(rec.name, sizeof(rec.name), stdin);
    printf("Salary: ");
    scanf("%d", &rec.salary);
    fwrite(&rec, sizeof(Record), 1, fp);
  }
  fclose(fp);
}

void show_file(const char *filename) {
  FILE  *fp;
  Record rec;
  int    i = 1;

  fp = fopen(filename, "rb");
  if(fp == NULL) {
    printf("Error!");
    exit(1);
  }

  printf("\nNO.\t SALARY \t FIRST AND LAST NAME\n");
  while(fread(&rec, sizeof(Record), 1, fp) > 0) {
    printf("%d \t %d \t \t %s", i, rec.salary, rec.name);
    i++;
  }
  fclose(fp);
}
```

Listing 9.2: Program L9Ex2.c

9.3 Lab Tasks

1. Run the programs given as examples and analyze the output.

2. Write code to accomplish each of the following:
   a) Open the file "file.txt" for writing (and creation) and assign the returned file pointer to file_ptr.
   b) Write a record to the file "file.txt". The record includes the integer account_num, the string name and the floating-point current_balance. The record values are in-

serted by the user from the keyboard.
c) Read a record from the file "file.txt" and put the values in the following variables: integer account_num1, string name1 and floating-point current_balance1.
d) Display the values held by the variables account_num1, name1 and current_balance1.
e) Close the file "file.txt".

3. Write a C program to replace the text of a specific line in a file with another text, and to delete a specific line from the file. The users of the program can perform these operations repeatedly until the '3' key is pressed. Use a temporary text file and the *remove* and *rename* built-in functions.

4. Write a C program to store information for a class of several students in a binary file. Also, the program must conduct various statistical analyses based on the data stored in the file. The users of the program can choose the statistical analysis from a menu.

9.4 References

1. Paul Deitel, Harvey Deitel, *C How to Program*, 2010, Sixth edition, Pearson Education, ISBN: 978-0-13-612356-9.

Laboratory paper 10

Embedded Systems Programming Case Study

10.1 Overview

- Presentation of Webots environment
- Use of C concepts learned so far to develop embedded systems in Webots
- Work time: 2 hours

10.2 Theoretical Considerations

Webots is an open-source, multi-platform desktop application which offers a comprehensive development environment for modeling, programming, and simulating robotic systems [1]. Webots allows anyone with basic knowledge in one of the languages, C, C++, Java, Python or MATLAB, to create 3D virtual worlds with physics properties where mobile robots with different locomotion schemes can be programmed to follow a specific behaviour.

A Webots simulation consists of the following components [2]:

- A Webots world file that defines one or more robots along with their environment;
- One or more controller programs for the robots;
- An optional physics plugin that can alter Webots' standard physics behavior.

The Webots world includes a detailed description of each object of the 3D representation, such as its position, orientation, geometry, appearance, physical properties, type of object. A Webots world is stored in a .wbt file, which is located in the "worlds" subdirectory of a Webots project. The world file doesn't contain the controller code of the robots, instead, it specifies the name of the controller needed for each robot. The controller is a computer program, written in one of the following programming languages C, C++, Java, Python or MATLAB, that controls a robot defined in a world file. The source file of a controller is located in the "controllers" subdirectory of a Webots project.

10.2.1 Webots installation and guided tour

Webots can be installed on Linux, Windows, and on macOs. The complete installation instructions are presented in [3] for each individual operating system previously mentioned.

The installation procedure on Windows is a simple one that involves downloading the installation kit for Windows (available at `https://cyberbotics.com/`), launching it into execution and following the installations steps. Before installation, Webots system requirements from `https://cyberbotics.com/doc/guide/system-requirements` should be checked. Webots R2023b version was installed and used to implement the Webots application from this laboratory paper.

After the installation is complete, Webots provides a quick overview of some robotics simulations through the Webots guided tour. An example of two robotic arms simulation running in Webots is presented in Figure 10.1. As can be seen in Figure 10.1, Webots environment has three main areas:

- The center area – provides 3D viewing of the simulation;
- The left area – enables the programmers to manage and configure the components of the simulation world;
- The right area – offers an integrated code editor for writing and compiling the controllers for the robots.

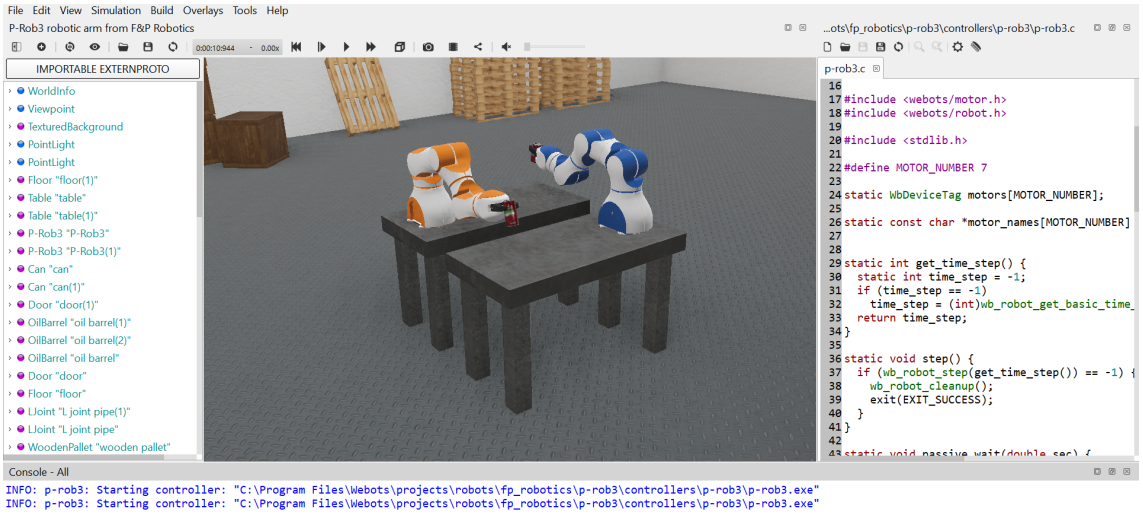


Figure 10.1: Robotic arms simulation running in Webots

10.2.2 Development of a Webots application

Webots allows the development of an application that uses a robotic arm to sort the luggage for different flights depending on a color tag, each color representing a different flight number. The entire process of the application development is described in several steps, as follows.

STEP 1: Launch the Webots environment and create a new project

After Webots is opened, from the File menu, select New − > New project directory. The graphical user interfaces presented in Figures 10.2, 10.3, 10.4 appear and the name of the project should be provided and also the name of the world file. Webots creates a directory having the name of the project and several subdirectories where worlds, controllers, libraries, plugins, and protos can be defined. In this example, the file luggage_sorter_robot.wbt is created in worlds folder.

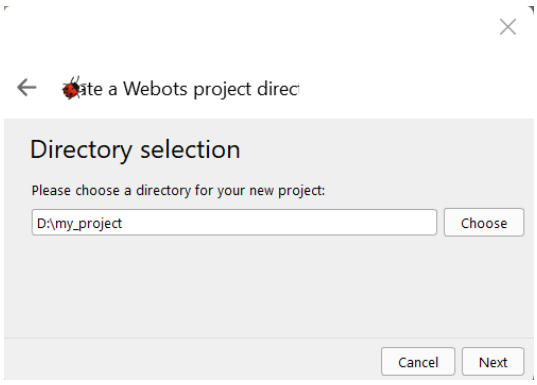


Figure 10.2: Webots graphical user interface to choose a directory for the new project

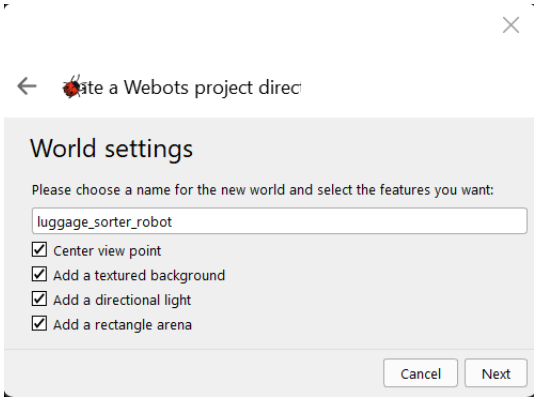


Figure 10.3: Webots graphical user interface to choose a name for the world file

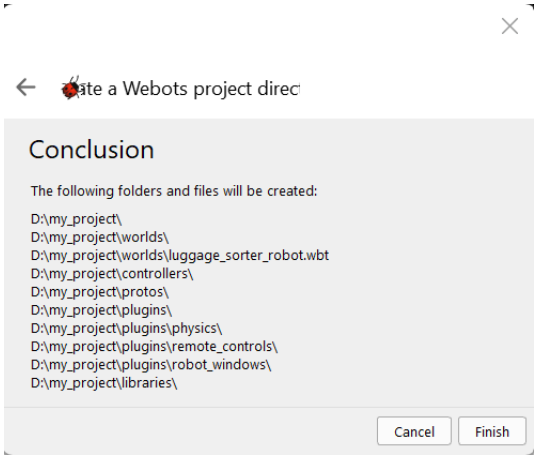


Figure 10.4: Summary of the folders and files generated

STEP 2: Create and customize the virtual word (i.e., rectangular area, objects, robots)

The Webots environment left area allows the customization of the rectangular area (type of floor, floor size, wall thickness) and of the texture background. Next, predefined objects, called PROTO nodes can be added in the simulation world on the rectangle area. For this application, four conveyor belts are added, one for all the luggage and three for separating the luggage according to a flight. The luggage are simulated using solid boxes customized with different photos to change their aspects. Also, the recognition colors property is set for these solid boxes in order to classify them for each flight. At the end of conveyor belts there are metal storage boxes, and advertising boards with the flights numbers are added to the conveyor belts as well. The robot chosen for this application is a robotic arm of type UR10e with a Robotiq3fGripper (https://webots.cloud/run?url=https://github.com/cyberbotics/webots/blob/released/projects/robots/universal_robots/protos/UR10e.proto). This robot has a camera attached, with a recognition node and a distance sensor node. The robot is placed on a solid box. The controller property of the robot makes the connection between the virtual world and the source code that ensures the behavior of the robot. Therefore, a source code file having the name exactly like the name of the controller property should be added to the project, as the next step shows.

All these objects can be easily added in the rectangular area using the Add a note dialog (Figure 10.5) that allows the selection of predefined PROTO nodes.

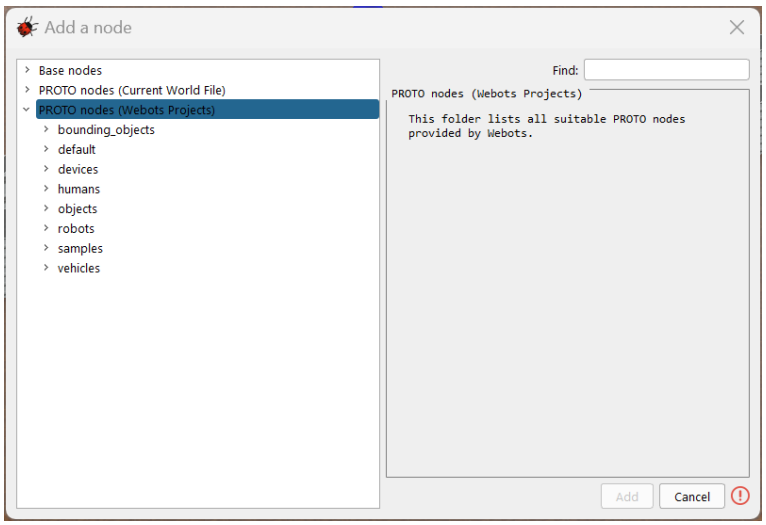


Figure 10.5: Add a node dialog

Figure 10.6 presents the virtual world created for this application.

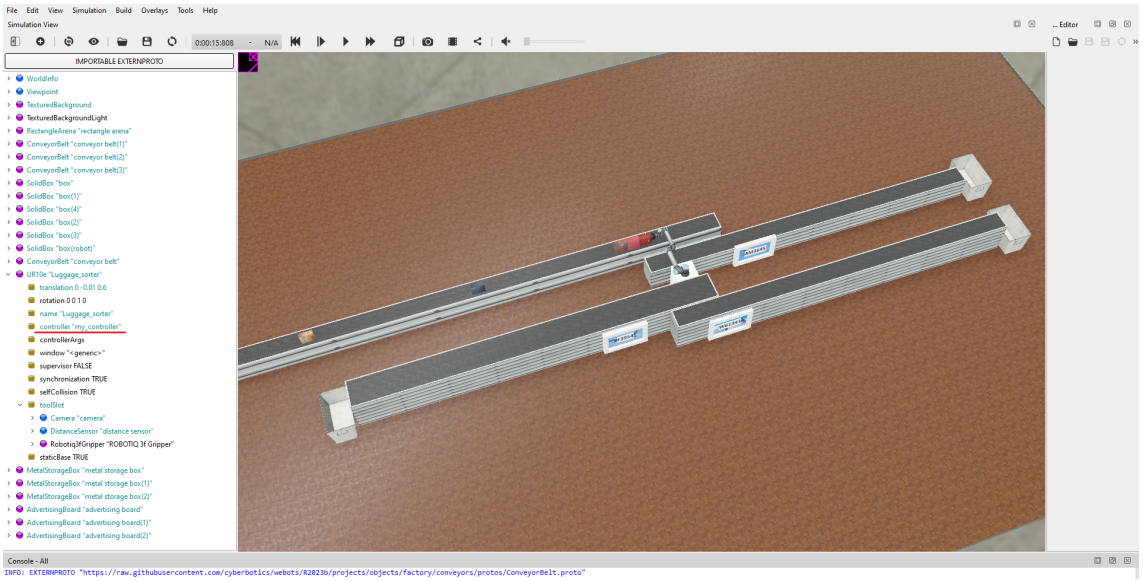


Figure 10.6: Virtual world after creating and positioning the objects (PROTO nodes)

STEP 3: Create a C source code file for the robot's controller

Next, to add the robot's controller file source, from the File menu, select New − > New Robot Controller. The graphical user interfaces presented in Figure 10.7 appear and the name of the programming language should be selected and the name of the controller should be provided. Webots creates a source file with the *main* function, where the next three functions from <webots/robot.h> are called: *wb_robot_init()* to initiliaze Webots

components, *wb_robot_step()* to perform the simulation steps, and *wb_robot_cleanup()* to cleanup Webots resources.

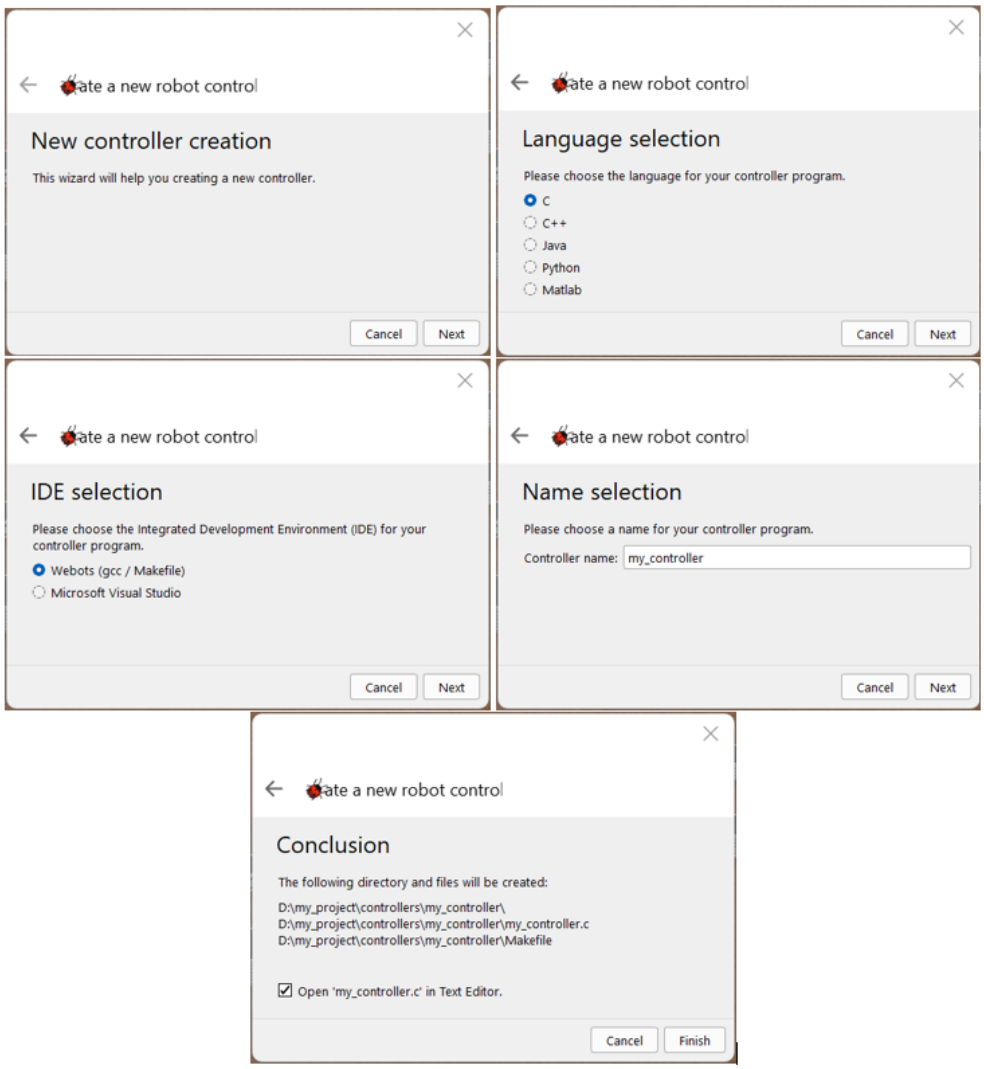


Figure 10.7: Webots graphical user interfaces to create a robot's controller

In order to implement the behaviour of the robot to sort the luggage for different flights, more lines of code should be added in the controller. Listing 10.1 presents the controller's code. It represents an adapted version of the controller from [4].

```
1  /* my_controller.c */
2
3  #include <webots/robot.h>
4  #include <webots/distance_sensor.h>
5  #include <webots/motor.h>
6  #include <webots/position_sensor.h>
7  #include <webots/robot.h>
8  #include <webots/camera.h>
9  #include <webots/camera_recognition_object.h>
10 #include <stdio.h>
11
12 #define TIME_STEP 32
13
```

```c
/* definition of an enumeration data type which represents the 5 stages of
    robot's movements */
enum State { WAITING, GRASPING, ROTATING, RELEASING, ROTATING_BACK };

int main() {
  /* initialization of Webots robot */
  wb_robot_init();

  /* definition and initialization of some variables */
  int i = 0, j = 0;
  int state = WAITING;

  /* definition and initialization of constants representing the positions
    of the joints */
  const double target_positions[] = {-1.3, -3, -2.38, -2};
  const double target_positions1[] = {-2.3, 2.8, -2.31, -1.51};
  const double target_positions2[] = {-2, -1.6, -2.5, -1.51};

  /* definition and initialization of variables representing the three
    colors used for sorting the luggage */
  double r = 0.00, g = 0.00, b = 0.00;

  /* definition and initialization of a variable for the robot's speed */
  double speed = 1.00;

  /* definition and enabling of the robot's camera and camera recognition */
  WbDeviceTag camera = wb_robot_get_device("camera");
  wb_camera_enable(camera, 2 * TIME_STEP);
  wb_camera_recognition_enable(camera, 2 * TIME_STEP);

  /* definition and initilization of an array for the gripper part of the
    robot */
  WbDeviceTag hand_motors[3];
  hand_motors[0] = wb_robot_get_device("finger_1_joint_1");
  hand_motors[1] = wb_robot_get_device("finger_2_joint_1");
  hand_motors[2] = wb_robot_get_device("finger_middle_joint_1");

  /* definition and initilization of an array for the robot's main body  */
  WbDeviceTag ur_motors[4];
  ur_motors[0] = wb_robot_get_device("shoulder_lift_joint");
  ur_motors[1] = wb_robot_get_device("elbow_joint");
  ur_motors[2] = wb_robot_get_device("wrist_1_joint");
  ur_motors[3] = wb_robot_get_device("wrist_2_joint");

  /* definition and enabling of the distance sensor attached to the robot */
  WbDeviceTag distance_sensor = wb_robot_get_device("distance sensor");
  wb_distance_sensor_enable(distance_sensor, TIME_STEP);

  /* definition and enabling of the position sensor attached to the robot */
  WbDeviceTag position_sensor = wb_robot_get_device("wrist_1_joint_sensor");
  wb_position_sensor_enable(position_sensor, TIME_STEP);

  /* setting the speed to the motors of the robot's main body */
  for(i = 0; i < 4; ++i)
    wb_motor_set_velocity(ur_motors[i], speed);

  /* setting the initial position of the robot */
  for(i = 0; i < 4; ++i)
    wb_motor_set_position(ur_motors[i], 0.0);
  for(i = 0; i < 3; ++i)
     wb_motor_set_position(hand_motors[i], 0.05);
```

```
71
72
73  /* main loop to perform simulation steps of TIME_STEP milliseconds and
      leave the loop when the simulation is over */
74  while(wb_robot_step(TIME_STEP) != -1) {
75    printf("start----------------------\n");
76
77    /* definition and initialization of the number of objects recognized by
      the robot's camera */
78    int number_of_objects = wb_camera_recognition_get_number_of_objects(
      camera);
79    printf("\nNumber of luggages: %d.\n", number_of_objects);
80
81    const WbCameraRecognitionObject *objects =
      wb_camera_recognition_get_objects(camera);
82
83    /* color extraction based on the solid box recognitionColors property */
84    for(i = 0; i < number_of_objects; ++i) {
85      for(j = 0; j < objects[i].number_of_colors; ++j) {
86        r = objects[i].colors[3 * j];
87        g = objects[i].colors[3 * j + 1];
88        b = objects[i].colors[3 * j + 2];
89        if (r == 1.000000 && g == 0.000000 && b == 0.000000)
90          printf("FLIGHT NUMBER: W61343\n");
91        else if(r == 0.000000 && g == 1.000000 && b == 0.000000)
92          printf("FLIGHT NUMBER: AM3645\n");
93        else if(r == 0.000000 && g == 0.000000 && b == 1.000000)
94            printf("FLIGHT NUMBER: F3954\n");
95        }
96    }
97
98    /* robot's behaviour implementation taken into account the states of the
       arm */
99    switch(state) {
100       case WAITING:
101       for(i = 0; i < 3; ++i)
102             wb_motor_set_position(hand_motors[i], 0.05);
103       printf("Distance: %lf\n", wb_distance_sensor_get_value(
      distance_sensor));
104       fflush(stdout);
105       if(wb_distance_sensor_get_value(distance_sensor) < 300 &&
      wb_distance_sensor_get_value(distance_sensor) > 100) {
106         state = GRASPING;
107         printf("Grasping object\n");
108         for(i = 0; i < 3; ++i)
109           wb_motor_set_position(hand_motors[i], 0.85);
110       }
111       break;
112       case GRASPING:
113       if(r == 1.000000 && g == 0.000000 && b == 0.000000)
114         for(int i = 0; i < 4; ++i)
115           wb_motor_set_position(ur_motors[i], target_positions2[i]);
116       else if (r == 0.000000 && g == 1.000000 && b == 0.000000)
117         for(i = 0; i < 4; ++i)
118           wb_motor_set_position(ur_motors[i], target_positions1[i]);
119       else if(r == 0.000000 && g == 0.000000 && b == 1.000000)
120         for(i = 0; i < 4; i++)
121           wb_motor_set_position(ur_motors[i], target_positions[i]);
122       printf("Rotating arm\n");
123       state = ROTATING;
124       break;
```

```
125         case ROTATING:
126         if(wb_position_sensor_get_value(position_sensor) < -2.3) {
127           printf("Releasing object\n");
128           state = RELEASING;
129           for(i = 0; i < 3; ++i)
130             wb_motor_set_position(hand_motors[i], wb_motor_get_min_position(
    hand_motors[i]));
131         }
132         break;
133         case RELEASING:
134         for(i = 0; i < 4; ++i)
135           wb_motor_set_position(ur_motors[i], 0.0);
136         printf("Rotating arm back\n");
137         state = ROTATING_BACK;
138         break;
139         case ROTATING_BACK:
140         if(wb_position_sensor_get_value(position_sensor) > -0.1) {
141           state = WAITING;
142           printf("Waiting object\n");
143         }
144         break;
145     }
146     printf("------------------------end\n");
147   }
148
149   /* cleaning up Webots resources */
150   wb_robot_cleanup();
151   return 0;
152 }
```

Listing 10.1: Program L10Ex1.c

To build the controller's code, select Build from the Build menu. This operation creates the connection between the robot in the simulation world and its behavior implemented in C. If if there are no errors reported in the building phase, then select Real-time from the Simulation menu to see the application running.

10.3 Lab Tasks

1. Install Webots environment on your computers.
2. Go through the instructions in section 10.2.2 step by step and create the application.

10.4 References

1. Webots, https://cyberbotics.com/#webots, Accessed in September 2024.
2. Webots User Guide, https://cyberbotics.com/doc/guide/introduction-to-webots, Accessed in September 2024.
3. Webots User Guide, https://cyberbotics.com/doc/guide/installation-procedure, Accessed in September 2024.
4. Sorting of objects based on their color using a robotic arm in WEBOTS Software, https://drive.google.com/file/d/1E8nTn6trffT3joGyED4TO6PPOKz5X4F_/view, Accessed in September 2024.

Appendix - Data Representation in Computer Memory

The computers stores data in binary form. Binary data are made up of binary digits (digits 0 and 1). A binary digit, or bit, is the smallest unit of data in computing. The hexadecimal (base 16) or octal (base 8) number systems are also used as a compact form for representing binary data.

The way in which various types of data (integers, floating-point numbers, characters) transmitted by programs are converted into binary data and represented in the computer's memory is presented in the following.

1 Integer Data Representation

In general, computers use a fixed number of bits to represent an integer (8 bits, 16 bits, 32 bits or 64 bits) and also the next two representation schemes:

- Unsigned integers – this scheme can represent zero and positive integers;
- Signed integers – this scheme can represent zero, positive and negative integers. Three representation schemes can be used for signed integers:
  - Sign-magnitude representation;
  - 1's complement representation;
  - 2's complement representation.

The C programming language uses 16 bits (2 bytes) to represent a short integer, and 32 bits (4 bytes) to represent an integer or a long integer. In the following, the aspects related to integers represented on 32 bits are discussed (the same schemes apply to short integers as well, the only difference being the length of the representation, 16 bits).

1.1 Unsigned integers representation in C

An unsigned decimal integer is converted in binary and represented according to the bit-length, 32 bits (zeros are added to the left to complete the bit-length of the representation).

Figure 1 presents an example of unsigned integer number binary representation in the computer's memory.

Unsigned integer number (Decimal)	Bit-length	Unsigned binary representation (Binary and Hexadecimal)
5	32-bit	00000000 00000000 00000000 00000101 00 00 00 05

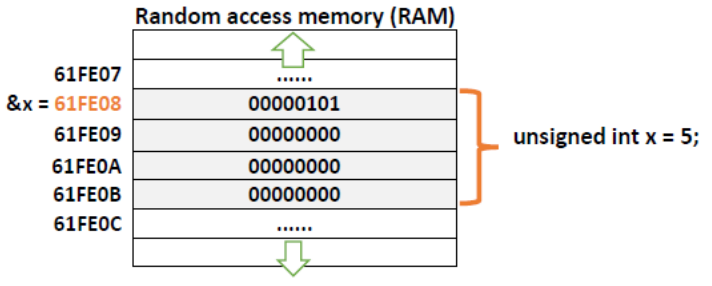


Figure 1: Example of an unsigned integer number representation

1.2 Signed integers representation in C

A decimal signed integer is represented using 2's complement scheme. This scheme avoids having two representations for 0, allows positive integer numbers and negative integer numbers to be treated in the same way, and more importantly, makes it easier to do arithmetic with negative integer numbers (the addition and subtraction are treated as one operation).

The 2's complement of positive integer numbers is obtained using the same method as in case of unsigned integers. The positive integer number is converted in binary and represented according to the bit-length, 32 bits (zeros are added to the left to complete the bit-length of the representation).

The 2's complement of negative integer numbers is obtained according to the next algorithm:

- Step 1. The absolute value of the negative integer number is converted in binary.
- Step 2. All the digits are inverted (0 becomes 1 and 1 becomes 0).
- Step 3. 1 is added to the result.

Figure 2 presents an example of signed integer number (negative) binary representation in the computer's memory.

Signed integer number (Decimal)	Bit-length	2's complement representation (Binary and Hexadecimal)
-5	32-bit	11111111 11111111 11111111 11111011 FF FF FF FB

1. The conversion of absolute value of -5 in binary is 00000000 00000000 00000000 00000101.
2. The binary value with the digits inverted is 11111111 11111111 11111111 11111010.
3. 1 is added to the result:

```
11111111 11111111 11111111 11111010 +
                                    1
------------------------------------------------------
11111111 11111111 11111111 11111011
```

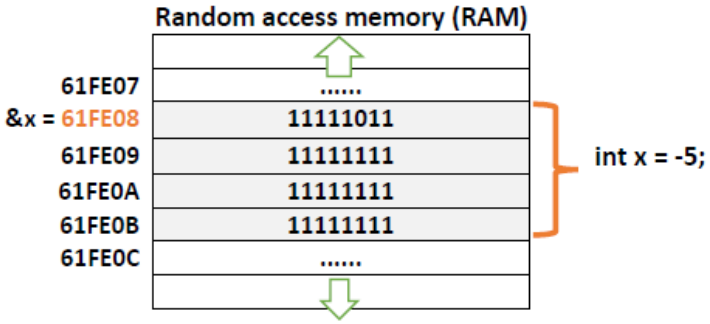


Figure 2: Example of a negative integer number representation

2 Floating-Point Data Representation

In computers, floating-point numbers are represented in the scientific notation, with a fraction (F), and an exponent (E) of a certain radix (r), in the form of $F * r^E$. F and E can

be positive or negative. Decimal numbers use radix of 10 ($F * 10^E$), while binary numbers use radix of 2 ($F * 2^E$).

Modern computers implement the IEEE 754 Standard for representing floating-point numbers. There are two representation schemes:

- 32-bit single-precision;
- 64-bit double-precision.

The C programming language uses a 32-bit single precision scheme to represent a float, and 64-bit double precision to represent a double. In the following, the aspects related to floats represented on 32 bits and doubles represented on 64 bits are discussed.

2.1 Floats representation in C

According to the IEEE 754 Standard the significance of the 32 bits of a float is as follows (Figure 3):

- The most significant bit (S) is the sign bit with 0 for positive float numbers and 1 for negative float numbers;
- The following 8 bits represent the biased exponent (e);
- The remaining 23 bits represent the fraction (F) (also called the mantissa or significand);
- Between the actual exponent (E) and the biased exponent (e) there is the formula: $E = e - 127$.

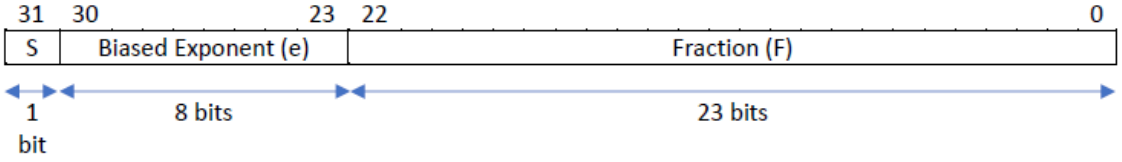


Figure 3: Floats representation

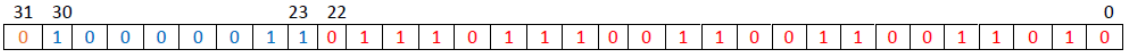
The 32-bit single precision representation of positive and negative floats numbers involves calculating the values for the sign bit (S), biased exponent (e) and fraction (F) according to the next algorithm:

- Step 1. The absolute value of the float number is converted in binary (both the integer part and the fractional part are converted in binary following the specific conversion rules).
- Step 2. The fractional part is normalized so that in the normalized form, there is only one non-zero digit to the left of the radix point. The exponent (E) is obtained in this step (the power of 2), as well as the fraction (F).
- Step 3. If the float is positive, then the sign bit (S) is 0, otherwise it is 1.
- Step 4. The biased exponent (e) is calculated based on the formula $e - 127 = E$. The result is converted in binary.
- Step 5. The number is represented using the values of S, e and F (previously calculated).

Figure 4 presents an example of a positive float number binary representation in the computer's memory.

Positive float number (Decimal)	Bit-length	Floating-point number representation (IEEE 754 Standard) (Binary and Hexadecimal)
23.45	32-bit	01000001 10111011 10011001 10011010 41 BB 99 9A

1. The conversion of the decimal number in binary is $10111.0111001100110011010_{(2)}$.
2. The normalized form of the binary number is $1.01110111001100110011010 * 2^4$ => E = 4, F = 01110111001100110011010.
3. The number is positive => S = 0.
4. The exponent E is 4, so in the biased form it is: e - 127 = 4 => $e = 131_{(10)} = 10000011_{(2)}$.
5. The fraction F is 01110111001100110011010 (looking to the right of the radix point in the normalized form).



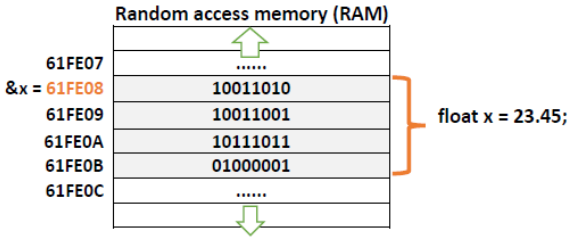


Figure 4: Example of a positive float number representation

2.2 Doubles representation in C

According to the IEEE 754 Standard the significance of the 64 bits of a double is as follows (Figure 5):

- The most significant bit (S) is the sign bit with 0 for positive double numbers and 1 for negative double numbers;
- The following 11 bits represent the biased exponent (e);
- The remaining 52 bits represent the fraction (F) (also called the mantissa or significand);
- Between the actual exponent (E) and the biased exponent (e) there is the formula: $E = e - 1023$.

Figure 5: Doubles representation

The 64-bit single precision representation of positive and negative double numbers involves calculating the values for the sign bit (S), biased exposent (e) and fraction (F) according to the next algorithm:

- Step 1. The absolute value of the double number is converted in binary (both the integer part and the fractional part are converted in binary following the specific conversion rules).
- Step 2. The fractional part is normalized so that in the normalized form, there is only one non-zero digit to the left of the radix point. The exponent (E) is obtained in this step (the power of 2), as well as the fraction (F).

– Step 3. If the double is positive, then the sign bit (S) is 0, otherwise it is 1.
– Step 4. The biased exponent (e) is calculated based on the formula $e - 1023 = E$. The result is converted in binary.
– Step 5. The number is represented using the values of S, e and F (previously calculated).

Figure 6 presents an example of a negative double number binary representation in the computer's memory.

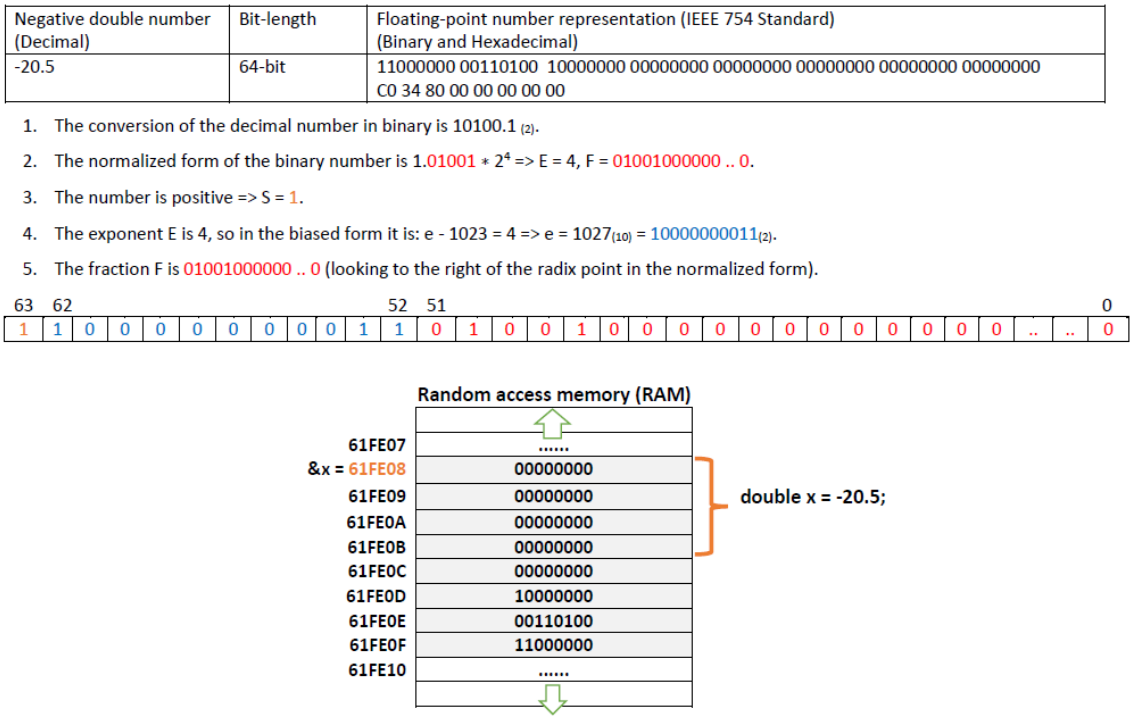


Figure 6: Example of a negative double number representation

3 Character Data Representation

Character data is composed of letters, symbols, and numerals that are not used in calculations. Character data is commonly referred to as text. In computer memory, character data are represented using several encoding schemes. Some of them are listed in the following:

– ASCII (American Standard Code for Information Interchange) requires 7 bits for each character. It provides codes for 128 characters;
– Extended ASCII extends ASCII and uses 8 bits for each character. Using 8 bits instead of 7 bits allows Extended ASCII to provide codes for 256 characters;
– Unicode uses 16 bits and provides codes for 65 000 characters. It can be used for representing the alphabets of multiple languages;
– UTF-8 (Unicode Transformation Format - 8-bit) is a variable-length coding scheme that uses 7 bits for common ASCII characters and 16 bits Unicode as necessary.

The C programming language uses 8 bits to represent a character. The ASCII/Extended ASCII scheme is used. The integer value of the character, according to the encoding

scheme, is stored in the computer's memory.

3.1 Unsigned characters representation in C

An unsigned character has values in the range from 0 to 255 and is represented using 8 bits in the computer's memory and Extended ASCII encoding scheme. The decimal value of the ASCII code of the character is converted in binary and represented according to the bit-length, 8 bits.

Figure 7 presents an example of an unsigned character binary representation in the computer's memory.

Unsigned character	ASCII code in decimal	Bit-length	Extended ASCII representation (Binary and Hexadecimal)
° (degree symbol)	176	8-bit	10110000 B0

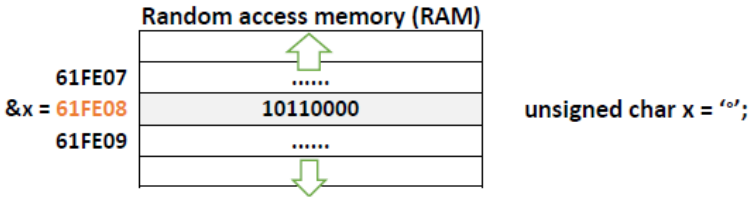


Figure 7: Example of an unsigned character representation

3.2 Signed characters representation in C

A signed character has values in range the range from -127 to 128 and is represented using 8 bits in the computer's memory and Extended ASCII encoding scheme. Since a signed character can have a negative value, its representation in binary follows the 2's complement scheme rules. The ASCII code of the character is converted in binary and represented according to the bit-length, 8 bits.

Figure 8 presents an example of a signed character binary representation in the computer's memory.

Signed character	ASCII code in decimal	Bit-length	Extended ASCII representation (Binary and Hexadecimal)
t	116	8-bit	01110100 74

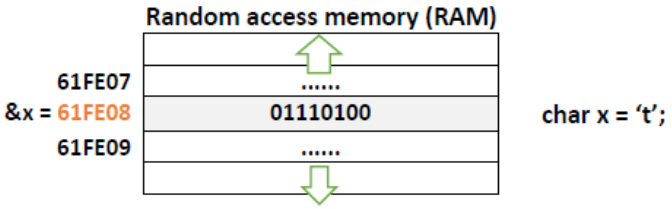


Figure 8: Example of a signed character representation

Bibliography

1. Two's Complement, https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html, Accessed in September 2024.
2. IEEE Computer Society, *IEEE Standard for Floating-Point Arithmetic*, https://iremi.univ-reunion.fr/IMG/pdf/ieee-754-2008.pdf, Accessed in September 2024.
3. ASCII Table, https://www.ascii-code.com/, Accessed in September 2024.