Computer Programming

Introduction. I/O functions

Robert Varga

Technical University of Cluj-Napoca
Computer Science Department

Course 1

Contents

Contents

Contact info

- Course and Seminar
  - Robert Varga
  - email: robert.varga@cs.utcluj.ro
  - website: link
- Laboratory
  - Vivian Chiciudean
  - Robert Varga
- MS Teams
  - Homework
  - Announcements
  - preferred means of communication

Grading

- Seminar
  - 3 verified homeworks
- Laboratory
  - Laboratory test - grade must be above 5
  - 40% of your final grade
- Course
  - be present at 10 out of 14 courses = 1 point
  - course quizzes = 1 point
  - Written exam - grade must be above 5
  - 60% of your final grade
- Opportunities to gain extra grade points

Course objectives

- learn programming in the C language:
  - general programming concepts and techniques - applicable to other languages
  - elements specific to the C language
  - usage of development tools
- skills acquired:
  - basic algorithm design
  - implementing an algorithm as a C program
  - analysis and understanding of written code
  - debugging

Recommended reading

- Books
  - Kim N. King - C programming: a Modern Approach
  - Brian Kernighan, Dennis Ritchie - The C Programming Language
- Problems
  - CP laboratory and homework
  - infoarena - problems solved by at least 500 users
  - atcoder - competitive programming platform
  - leetcode - technical interview questions
- Official language documentation
  - cppreference

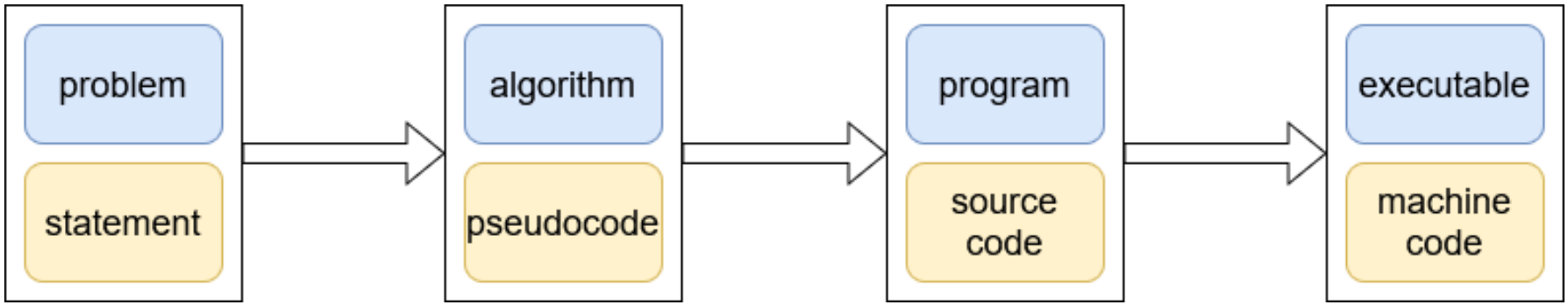
Course contents

- Fundamental C syntax
  - Introduction. I/O Functions
  - Data Types. Expressions. Conditional Statements
  - Bitwise Operations. Repetitive Statements
  - Functions. Arrays
- Higher level elements
  - Preprocessing Directives
  - Pointers 1
  - Pointers 2
  - Strings
  - Recursion
  - Structures
  - Files
  - Command Line Arguments

Contents

Definitions

- a computational problem is a description of desired output values for a set of given input values
- an algorithm specifies the required steps to transform input data to obtain the output data
- a program is the implementation of an algorithm in a programming language



Algorithm

- it defines the required steps to transform the input data into the output data
- often described in a semi-formal language called pseudocode
- characteristics:
  - correct - on all possible inputs
  - finite - it requires finite number of steps to finish
  - efficient - obtains the result using the minimal number of steps
  - economic - uses the least amount of memory

Algorithm 1.1 - example

```
read x
p = 1
while p < x do
    while x is divisible by p do
        print p
        x = x / p
    end while
    p = p + 1
end while
```

- what does it achieve?
- is it correct?
- is it guaranteed to finish in finite steps?
- is it the most efficient?
- is it economic?

History of the C language

- developed by Dennis Ritchie at Bell Labs in the 1970s
- related to the Unix operating system
- used in scenarios where speed and efficiency is important (real-time systems, embedded systems)
- characteristics:
  - imperative - it is based on commands/instructions
  - compiled - the source code is translated in machine code using a compiler
  - high-level - abstractization from the hardware
  - short - uses relatively few keywords

Contents

Program 1.1 - Hello World

```
1 // My first C program
2 #include <stdio.h>
3 int main(void)
4 {
5   printf("Hello, World!");
6   return 0;
7 }
```

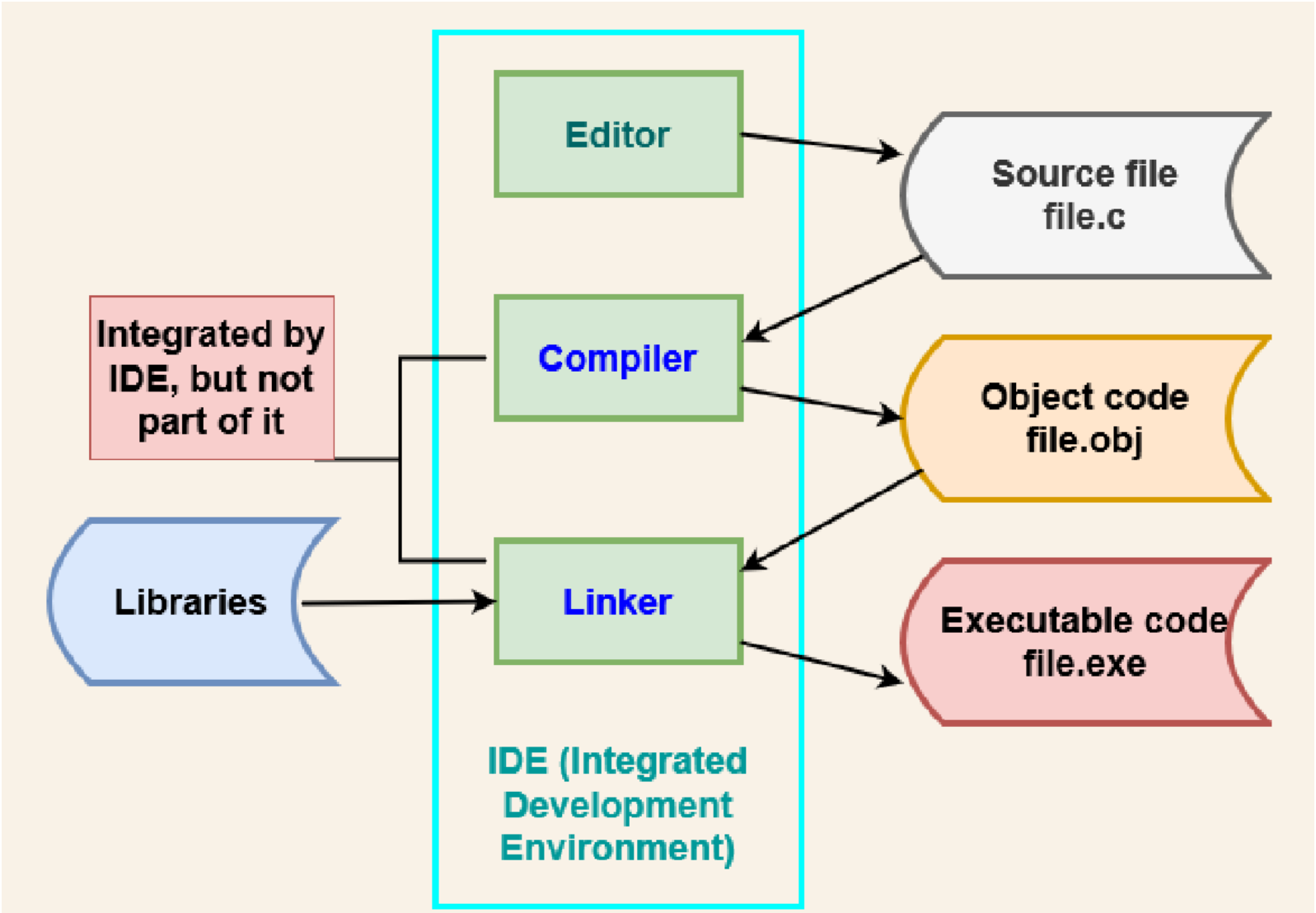
- first line is comment, does not affect the behavior
- second line includes the standard library header, needed for the print operation
- the main function starting from line 3 defines the starting point of the program

Program 1.1 - Hello World

```c
// My first C program
#include <stdio.h>
int main(void)
{
  printf("Hello, World!");
  return 0;
}
```

- the printf function prints the string on the screen
- the return instruction terminates the main function and with it the whole program
- code blocks are delimited by { } pairs

From code to executable



Contents

General properties

- all statements are terminated by semicolon ;
- case sensitive - lower case and upper case letters are different
- indentation can and should be used to indicate the structure of the program
- whitespace characters (space, tabs, new-line) are ignored by the compiler

Variables

- a **variable** is used to store useful information during program execution
- the name reflects that it can be changed
- analogous to variables or unknowns from mathematics
- the C language requires us to **declare** all variables
- declaration specifies the type and name, in this order
- you can declare multiple variables of the same type by putting commas between their names

```
1 char c;
2 int i, j, k;
3 float temp_celsius, temp_fahrenheit;
```

Naming

Variable names need to follow the rules:

- cannot start with numbers
- starting with single or double underscore is not recommended
- cannot contain spaces
- words can be linked with underscore: my_temp_variable
- or we can use camel-case: myTempVariable
- recommended to be short but descriptive

Data Types

- the **data type** of a variable specifies its domain and the possible operations which can be performed on it
- the domain refers to the set of possible values it can take
- a key element in programming is choosing the appropriate data type
- for now, we only introduce the fundamental data types:
  - char - character - stores a symbol (letter, digit, whitespace)
  - int - integer - common type for small integers
  - float - single precision floating point number - numbers with decimal part
  - double - double precision floating point numbers
  - void - absence of type

Initialization and assignment

- **initialization** specifies starting values for variables
- variables can and should be initialized at the moment of their declaration
- in general, the C language does not guarantee initialization with 0 or any other value

```
1 char c = 'a';
2 int i = 1, j = -2, k = 3;
3 float x = 1.5, y = -.4;
```

- **assignment** refers to the operation of changing the value of a variable
- this happens after its declaration

```
1 char c;
2 c = 'a';
3 int i = 1;
4 i = 2;
```

I/O in C

- the C language does not have built-in instructions for reading and writing data
- these operations are handled by functions from the standard library stdio.h
- we will focus on the two main functions:
  - printf - formatted printing - prints on the screen = standard output stdout
  - scanf - formatted reading - reads from the keyboard = standard input stdin
- there is an additional output stream called stderr

Contents

printf function

int printf(const char* format, ...);

- prints on the screen using the specified format
- the format string is followed by 0 or more arguments
- input parameters
  - format - the format string specifies how the printing should be carried out
  - ... - additional parameters that hold what should be printed
- output value
  - returns the number of characters that were successfully written

printf function - simple examples

```
1  //Example 1
2  printf("hello");
3
4
5
6  //Example 2
7  int x = 7;
8  printf("%d", x);
9
10
11
12 //Example 3
13 printf("%c", 65);
```

- prints the given string as is
- no format specifier

- prints the variable x as a decimal number
- we have a single format specifier that dictates how the variable should be shown

- prints the number 65 interpreted as a character
- we have a single format specifier and a literal which is a number

Format specifier

- the general form of the specifier includes:

  % indicator dimension precision modifier conversion

- only the percent sign and the conversion are mandatory
- the number of format specifiers should be equal to the number of additional arguments supplied to the function

Format specifier - important ones

format specifier	**data type**	**explanation**
%c	char	writes a single character corresponding to the provided ASCII value
%d	int	writes an integer in base 10 (decimal)
%f	float, double	writes a floating point number in base 10
%s	char*	writes a string

Format specifier - extended list

format specifier	**explanation**
%o	interpret integer in base 8 (octal)
%x %X	interpret integer in base 16 (hexadecimal)
%u	interpret integer as unsigned
%e %E	interpret floating point number in scientific form
%g	interpret floating point number in short notation
%a %A	interpret floating point number in base 16
%p	interpret a memory address in base 16
%n	store the number of characters printed up to this symbol
%%	print the symbol %

- the lower-case and upper-case versions use lower-case or upper-case characters respectively

Format specifier - indicator

% **indicator** dimension precision modifier conversion

- it is optional
- changes the alignment of the data
- − align the data to the left
- + write the plus sign in front of positive numbers
- # write the prefix 0 for numbers in octal, the prefix 0x for numbers in hexadecimal and the decimal point for floating point numbers
- 0 complete the number with leading zeros (it is ignored if − is present)

Format specifier - dimension, precision

% indicator **dimension precision** modifier conversion

- dimension
  - it is an integer or the * symbol
  - if asterisk is used it takes the previously stored value
  - specifies the minimal width used for printing
  - all characters are printed if this is exceeded
  - default alignment is to the right
- precision
  - it is of the form .x or .*, where x is an integer
  - if asterisk is used it takes the previously stored value
  - specifies the number of digits to print after the decimal point

Format specifier - modifier

% indicator dimension precision **modifier** conversion

- changes the size of the data
- can be h (half), l (long) or L (long double)

%hhd	char	one byte
%hd	short	two bytes
%lld	long long int	eight bytes
%Lf	long double	ten bytes

printf function - complex example 1

```
1 printf("%-13.3f<<", 1.0/7);
```

- prints the decimal digits of the number $1/7$
- followed by the characters $<<$
- alignment is to the left
- use a field of size 13 to print
- print only 3 digits after the decimal point (rounding)

printf function - complex example 2

```
1 long long int x = 100000;
2 x = x * x;
3 printf("%+*lldxx", 20, x);
```

- construct the number $(10^5)^2$ in x
- print the digits of x
- followed by the characters xx
- default alignment is to the right
- use a field of size 20 to print, obtained after replacing asterisk with the value of the first argument
- show a plus sign in front of positive numbers

Escape sequence

- the backslash character has a special role in strings
- it introduces an escape sequence
- characters following it are printed differently
- $\backslash n$ signifies new line (enter)
- $\backslash b$ signifies backspace
- $\backslash t$ signifies tab
- $\backslash a$ issues an alert sound
- $\backslash$" prints quotation marks inside the string
- $\backslash\backslash$ prints backslash

puts function

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

- prints the string s on the screen
- puts a new-line character after the string
- input parameters:
  - s - the string to be printed
- output value
  - returns a non-negative number in case of success and EOF (-1) in case of failure

Contents

scanf function

int scanf(const char* format, ...);

- reads from the keyboard using the specified format
- the format string is followed by 0 or more arguments
- input parameters:
  - format - the format string specifying how the data is interpreted for reading
  - ... - additional parameters that hold the *addresses* where the data read should be written
- output value
  - returns the number values which were correctly read or EOF (-1) in case of a read error

scanf function - simple examples

```
1  //Example 1
2  int x;
3  scanf("%d", &x);
4
5
6
7  //Example 2
8  char c = 'a';
9  scanf("%c", &c);
10
11
12
13 //Example 3
14 char s[100];
15 scanf("%s", s);
```

- read a number in base 10 (decimal)
- store it in the variable x
- note, x is initialized before the read operation

- reads a single character
- store it in the variable c
- the previous value from c is overwritten in case of a successful read

- declare an array of characters = string
- read the whole string

scanf function - format specifier

- uses the same format specifiers as printf
- exception: double type requires %lf instead of %f
- dimension here dictates the number of characters read
- if the input contains a character which does not match the specifier, reading is terminated
- all unread characters remain in the *input buffer*
- additional characters in the format string that are not part of any format specifier must appear in the input
- space in the format string matches with any whitespace character from the input

scanf function - complex example 1

```
1 int a, b;
2 scanf("%d/%d", &a, &b);
```

- reads a fraction given in the form a/b
- the numbers must be next to the / symbol
- the two components are correctly extracted and saved in variables a and b

scanf function - complex example 2

```
1 int a, b;
2 scanf("%d / %d", &a, &b);
```

- reads a fraction given in the form a/b
- because of the additional space in the format string there can be 0 or more spaces between the numbers and /
- all these inputs are accepted: 1/2, 1 /2, 1/ 2, 1 / 2

scanf function - complex example 3

```
1 char s[100];
2 scanf("%[^\n]", s);
```

- use a format specifier based on a regex which reads until the newline character
- this is needed because the default format specifier for strings $\%s$ stops at the first whitespace
- notice, strings are different from simply data types, and no & is required before the variable name
- [*abc*] reads characters matching the ones from the bracket
- [^*abc*] reads characters different from the ones from the bracket

Important distinctions between printf/scanf

- all variables in scanf should be preceded by the symbol &
  - this returns the address of the variable
  - more details in the course about pointers
  - omitting the symbol leads to run-time errors
  - for string-type variables the symbol should not be used
- for basic read operations the format string for scanf should contain only the format specifiers
- double type uses the %lf format specifier for scanf

gets function

int gets(char* s);

- reads a string from the keyboard until new-line is typed
- it is saved in the provided string variable s
- the new-line character is not saved in the string s
- input parameters:
  - s - the string s which will store the data
- output value
  - returns s in case of success and the NULL pointer in case of an error
- useful for reading strings containing spaces
- remark: this version of the function is not secure because it does not protect against reading longer strings than the size of s

Common beginner I/O mistakes

```c
//Example 1
char c;
scanf("%c", c);


//Example 2
int a = 42;
printf("%f", a);


//Example 3
printf("%d");
```

- missing & operator in front of the variable
- wrong format specifier
- number of variables is not equal to the number of format specifiers

Program 1.2 - Basic I/O

```
1  #include <stdio.h>
2
3  int main(){
4    char name[100];
5    puts("What's your name?");
6    gets(name);
7    printf("Hello, %s!\n", name);
8    int age;
9    puts("Your age?");
10   scanf("%d", &age);
11   printf("Your age in base 8 is
       %o\n", age);
12   return 0;
13 }
```

- declare a variable of string type
- read with gets because it can contain spaces
- use %*o* format specifier to output in base 8