

# Introduction to C Programming

Jan Faigl

Department of Computer Science  
Faculty of Electrical Engineering  
Czech Technical University in Prague

Lecture 01

B3B36PRG – Programming in C

## Overview of the Lecture

- Part 1 – Course Organization
  - Course Organization
  - Course Goals and Means of Achieving the Course Goals
- Part 2 – Introduction to C Programming
  - Program in C
  - Values and Variables
  - Standard Input/Output

*K. N. King: chapters 1, 2, and 3*

# Part I Part 1 – Course Organization

## Course and Lecturer

### B3B36PRG – Programming in C

- Course web page <https://cw.fel.cvut.cz/wiki/courses/b3b36prg>
- Submission of the homework – **BRUTE** Upload System <https://cw.felk.cvut.cz/brute> and individually during the labs.
- Lecturer:

prof. Ing. **Jan Faigl**, Ph.D.



- Department of Computer Science – <http://cs.fel.cvut.cz>
- Artificial Intelligence Center (AIC) <http://aic.fel.cvut.cz>
- Center for Robotics and Autonomous Systems (CRAS) <http://robotics.fel.cvut.cz>
- Computational Robotics Laboratory (ComRob) <http://comrob.fel.cvut.cz>

## Course Organization

- B3B36PRG – Programming in C; Completion: Z,ZK; Credits: 6  
*Z – ungraded assessment, ZK – exam*
- 1 ECTS credit is about 25–30 hours per semester, six credits is about **150–180 hours per semester**.
  - Contact part (lecture and labs): 3 hours per week, 42 hours in total that is only about **25% of the expected effort** supposed to be dedicated to the course.
  - Exam including preparation: *10 hours*.
  - Home preparation (first **book reading** and followed by homework) approx **9 hours per week** *Median load*.
- Ongoing work during the semester
  - Homeworks *mandatory, optional, and bonus parts.*
  - Semestral project** – multi-thread computational applications.
- Exam test and implementation exam – verification of the acquired knowledge and skills from the teaching part of the semester. *An independent work with the computer in the lab (class room).*
- Attendance to labs, submission of homework, and semestral project.
- Consultation** - If you do not know, or spent too much time with the homework, consult with the instructor/lecturer.
  - Maximize the contact time during labs and lectures, ask questions, and discuss.**

## Course Evaluation

Point Source	Maximum Points	Required Minimum Points
Assignment	25	} <b>25</b>
Bonus Assignment	10	
Labs (MCU)	6	
Labs Quizes	4	
Semester project	30	<b>10</b>
Exam test	20	} <b>10</b>
Implementation exam	20	
<b>Total</b>	<b>115</b>	<b>55</b>

<sup>1</sup>If you fail the implementation and score exam test for 13 or more points, the following exam term is only for the implementation, and vice versa, if you do not ask otherwise. 55 points is solid E, not borderline, but solid. The exam test (and implementation) is not corrected but evaluated, the scoring is upper bound, i.e., it might contain less points than evaluated.

- The course can be passed with **ungraded assessment and exam**.
- All homework must be submitted and they have to pass the mandatory assessment.**

## Resources and Literature

### Textbook

„C Programming: A Modern Approach“ (King, 2008)

C Programming: A Modern Approach, 2nd Edition, K. N. King, W. W. Norton & Company, 2008, ISBN 860-1406428577



*The main course textbook*

### During the first weeks, take your time and read the book!

*The first homework HW01 deadline is on 15.03.2025!*

### Lectures – support for the textbook, slides, comments, and **your notes**.

*Demonstration source codes are provided as a part of the lecture materials!*

### Laboratory exercises – gain practical skills by doing homework (yourself).

## Further Books

Programming in C, 4th Edition, Stephen G. Kochan, Addison-Wesley, 2014, ISBN 978-0321776419

21st Century C: C Tips from the New School, Ben Klemens, O'Reilly Media, 2012, ISBN 978-1449327149

The C Programming Language, 2nd Edition (ANSI C), Brian W. Kernighan, Dennis M. Ritchie, Prentice Hall, 1988 (1st edition – 1978)

Advanced Programming in the UNIX Environment, 3rd edition, W. Richard Stevens, Stephen A. Rago Addison-Wesley, 2013, ISBN 978-0-321-63773-4

## Further Resources

The C++ Programming Language, 4th Edition (C++11), Bjarne Stroustrup, Addison-Wesley, 2013, ISBN 978-0321563842



Introduction to Algorithms, 3rd Edition, Cormen, Leiserson, Rivest, and Stein, The MIT Press, 2009, ISBN 978-0262033848



Algorithms, 4th Edition, Robert Sedgewick, Kevin Wayne, Addison-Wesley, 2011, ISBN 978-0321573513



Course Organization Course Goals and Means of Achieving the Course Goals

## Course Goals

- Master (yourself) programming skills. *Labs, homework, exam*
- Acquire knowledge of C programming language.
- Acquire experience of C programming to use it efficiently. *Your own experience!*
- Gain experience to read, write, and understand small C programs.
- Acquire programming habits to write
  - easy to read and understandable source codes
  - reusable programs.
- Experience programming with
  - Workstation/desktop computers – using services of operating system. *E.g., system calls, read/write files, input and outputs.*
  - Multithreaded applications.
  - Embedded applications – STM32F446 Nucleo.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 12 / 66

Course Organization Course Goals and Means of Achieving the Course Goals

## Teaching Programming in B3B36PRG

- Our aim is to build your experience and develop your programming skills.
  - Programming vs. algorithmization;
  - Programming is the "craft" of how to implement an algorithm correctly.
  - Functional is not enough - the program must be correct tool. *Expected input vs. what the user can input.*
- The learning load is therefore spread over the course of the semester.
  - Practice assignments and homework deadlines.
- Systematic development of programming skills throughout the semester is essential. *Typically, there is time at the beginning of the semester to understand the principles (reading the textbook)!*
- Without knowing the constructs and basic commands, you cannot program effectively.
- Know and know how to use (not "stick"). *Dependence on whisperer or Co-pilot!*
  - Starting with relatively simple tasks to learn programming constructs and how to organize source code. *Code clarity and the ability to navigate code efficiently!*
  - The assignments can be implemented based on the topics covered in the lectures/labs. *Solutions with more advanced constructs may be more elegant(shorter), but may not provide the necessary insight.*
- In the first lectures we cover the necessary knowledge, which is further deepened.
  - Exercises complement the lectures and give more space for practical learning.
- You can choose a practical way of absorbing programming knowledge from examples, which is suitable to complement theoretical preparation from textbook(s).

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 13 / 66

Course Organization Course Goals and Means of Achieving the Course Goals

## Overview of the Lectures

- Course information, Introduction to C programming *K. N. King: chapters 1, 2, and 3*
- Writing your program in C, control structures (loops), expressions *K. N. King: chapters 4, 5, 6, and 20*
- Data types, arrays, pointer, memory storage classes, function call *K. N. King: chapters 7, 8, 9, 10, 11, and 18*
- Data types: arrays, strings, and pointers *K. N. King: chapters 8, 11, 12, 13, and 17*
- Data types: Struct, Union, Enum, Bit fields. Preprocessor and Large Programs *K. N. King: chapters 10, 14, 15, 16, and 20*
- Input/Output – reading/writing from/to files and other communication channels, Standard C library – selected functions *K. N. King: chapters 21, 22, 23, 24, 26, and 27*
- Parallel and multi-thread programming – methods and synchronizations primitives
- Multi-thread application models, POSIX threads and C11 threads
- C programming language wrap up, examples such as linked lists
- Accuracy and Speed of Calculation
- ANSI C, C99, C11 and differences between C and C++ Introduction to C++.
- Reserve (Thursday's schedule)
- Quick introduction to C++
- Resource Ownership in C++

All supporting materials for the lectures are available at <https://cw.fel.cvut.cz/wiki/courses/b3b36prg/start>  
Read slides, textbook, or even watch the recorded lectures before the lecture contact time!

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 14 / 66

Course Organization Course Goals and Means of Achieving the Course Goals

## Homeworks

- 1+7 homework - seven for the workstation. <https://cw.fel.cvut.cz/wiki/courses/b3b36prg/hw/start>

- HW 00 – Testing (1 point) 1 h
- HW 01 – ASCII Art (2 points) 3 h  
*Coding style penalization – up to -100% from the gain points.*
- HW 02 – Prime Factorization (2 points + 4 points bonus) Coding style 4 h + 4 h (bonus)
- HW 03 – Caesar Cipher (2 points + 2 points bonus) Coding style 3 h + 3 h (bonus)
- HW 04 – Text Search (2 points + 3 points optional) 5 h
- HW 05 – Matrix Calculator (2 points + 3 points optional + 4 points bonus) Coding style! 6 h + 5 h (bonus)
- HW 06 – Circular Buffer (2 points + 2 points optional) 5 h
- HW 07 – Linked List Queue with Priorities (2 pts + 2 pts optional) 7 h
  - All homework must be submitted to award an ungraded assessment *Total about 42–47 hours. Late submission is penalized!*

- Coding style needs to be learned, penalization is to motivate you thinking about it and learn the craft of coding. *If you improve over the semester, penalization can be compensated at the end.*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 15 / 66

Course Organization Course Goals and Means of Achieving the Course Goals

## Semestral Project

- A combination of control and computational applications with multithreading, communication, and user interaction. <https://cw.fel.cvut.cz/wiki/courses/b3b36prg/semestral-project/start>
- Mandatory tasks can be awarded up to 20 points.
- Bonus part can be awarded for an additional 10 points. *Up to 30 points in the total for the semestral project.*
- Minimum required points: 10!  
*Deadline – best before 16.05.2025@23:59 CEST. Further updates and additional points might be possible!*
- Deadline – 18.05.2025@23:59 CEST.*
- The expected required time to finish the semestral project is about 30–50 hours.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 16 / 66

Course Organization Course Goals and Means of Achieving the Course Goals

## Expected and Reported Times Needed to Complete Homeworks

- B3B36PRG - Average sum of the reported median times.
  - 97 hours (with HW05B ~ 5 h, SEM ~ 49 h).
- 6 credits is about 150–180 hours that is
  - 42 h contact part
  - 10 h exam, and
  - about 100–128 hours for homework.
- Plan your work! Use the first weeks to read the textbook!

Reported (in the literature) programming courses success rate is about 30%–75%. It is usually at the end of other STEM courses. PRG is not an exception. 2023/2024: 69% (100% of awarded credits) 2022/2023: 73% (97% of awarded credits) 2021/2022: 60% (97% of awarded credits) 2020/2021: 60% (95% of awarded credits)

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 17 / 66

Course Organization Course Goals and Means of Achieving the Course Goals

## Homework Assignment – BRUTE

- BRUTE – Bundle for Reservation, Uploading, Testing and Evaluation
  - Formal check – compiling the program.
  - Functionality and correctness testing – checking output for a given input.
    - Public inputs and corresponding outputs / non-public inputs.
  - Test the program yourself before uploading it.
    - Using the available inputs and outputs.
    - Creating your own inputs and debugging the program.
    - Creating inputs with the included input generator.
    - Verifying the output with the attached test or reference program.
- Understanding the code and checking possible states.
  - For each line, you should be able to answer why it is there and what it does!
  - For each function or input retrieval from the user, parse the possible input values or function return values!
    - If the input or return value is critical in terms of functionality, check the input and/or the appropriate action, e.g., output a message and exit the program.

For example, the expected input is a number, and the user enters something else.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 18 / 66

Course Organization Course Goals and Means of Achieving the Course Goals

## Tasks and BRUTE

- Tasks are not just about submitting an implementation that passes the BRUTE tests.
  - The goal is not to submit tasks in BRUTE; it is to verify the program functionality.
  - BRUTE is a tool to continuously check your progress and gain knowledge.
  - The goal is to learn to independently program functional programs correctly.
- Tasks are all about gaining gradual experience with specific constructs.
  - All of the task assignments have been implemented many times, and even generative AI can do it.
  - In this course, you have the opportunity to understand C programming through your own implementation of assignments. The task successful submission is a means to reach the goal, not the goal itself.
- Tasks are very similar in relative difficulty. It is important to solve the tasks independently and to learn the sub-skills. *Absolutely, the tasks get progressively more and more difficult!*
- Rather than struggling too long on your own, ask (on Discord), for practice or consultation.
- Tasks HW01–HW03 and HW05 are checked for correctness and code clarity.
  - Focused on consistency, readability, and modularity (splitting into functions). *In terms of training and learning, try to split even a seemingly trivial program into multiple functions.*
  - The motivation is not to spend too much time with coding without significant progress.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 19 / 66

Program in C Values and Variables Standard Input/Output

## Part II

### Part 2 – Introduction to C Programming

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 20 / 66

Program in C Values and Variables Standard Input/Output

## C Programming Language

- Low-level programming language.
- System programming language (operating system).
  - Language for (embedded) systems — MCU, cross-compilation.
- A user (programmer) can do almost everything.
  - Initialization of the variables, release of the dynamically allocated memory, etc.
- Very close to the hardware resources of the computer.
  - Direct calls of OS services, direct access to registers and ports.
- Dealing with memory is crucial for correct behaviour of the program.
  - One of the goals of the PRG course is to acquire fundamental principles that can be further generalized for other programming languages. The C programming language provides great opportunity to become familiar with the memory model and key elements for writing efficient programs.

**It is highly recommended to have compilation of your program fully under control.**

*It may look difficult at the beginning, but it is relatively easy and straightforward. Therefore, we highly recommend to use fundamental tools for your program compilation. After you acquire basic skills, you can profit from them also in more complex development environments.*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 22 / 66

Program in C Values and Variables Standard Input/Output

## Writing Identifiers in C

- Identifiers are names of variables (custom types and functions).
  - Types and functions, viz further lectures.
- Rules for the identifiers
  - Characters a–z, A–Z, 0–9, and \_.
  - The first character is not a numeral.
  - Case sensitive.
  - Length of the identifier is not limited.
    - First 31 characters are significant – depends on the implementation / compiler.
- Keywords<sub>32</sub>
  - [auto](#) [break](#) [case](#) [char](#) [const](#) [continue](#) [default](#) [do](#) [double](#) [else](#) [enum](#) [extern](#) [float](#) [goto](#) [if](#) [int](#) [long](#) [register](#) [return](#) [short](#) [signed](#) [sizeof](#) [static](#) [struct](#) [switch](#) [typedef](#) [union](#) [unsigned](#) [void](#) [volatile](#) [while](#)

C98

C99 introduces, e.g., `inline`, `restrict`, `_Bool`, `_Complex`, `_Imaginary`.  
 C11 further adds, e.g., `_Alignas`, `_Alignof`, `_Atomic`, `_Generic`, `_Static_assert`, `_Thread_local`.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 25 / 66

Program in C Values and Variables Standard Input/Output

## Program Building: Compiling and Linking

- The previous example combines three particular steps of the program building in a single call of the command (`clang` or `gcc`).
- The particular steps can be performed individually.
  - Text preprocessing by the **preprocessor**, which utilizes its own macro language (commands with the prefix `#`).
    - All referenced header files are included into a single source file.
  - Compilation of the source file into the object file.
    - Names of the object files usually have the suffix `.o`.

```
clang -c program.c -o program.o
```

    - The command combines preprocessor and compiler.
  - Executable file is linked from the particular object files and referenced libraries by the linker (linking), e.g.,
 

```
clang program.o -o program
```

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 28 / 66

Program in C Values and Variables Standard Input/Output

## Writing Your C Program

- Source code of the C program is written in **text files**.
  - Header files usually with the suffix `.h`.
  - Sources files usually named with the suffix `.c`.
- Header and source files together with **declaration** and **definition** (of functions) support.
  - Organization** of sources into several files (modules) and libraries.
  - Modularity** – Header file declares a visible interface to others.
    - A description (list) of functions and their arguments without particular implementation.
  - Reusability**
    - Only the “interface” declared in the header files is needed to use functions from available binary libraries.
- Sources consists of **keywords**, language **constructs** such as **expressions** and programmer's **identifiers**:
  - variables** – named mamory space;
  - function names** – named sequences of instructions).

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 23 / 66

Program in C Values and Variables Standard Input/Output

## Simple C Program

```
1 #include <stdio.h>
2 int main(void)
3 {
4     puts("I like B3B36PRG!\n");
5     return 0;
6 }
```

lec01/program.c

Is the program correct? Or, strictly correct? See further!

- Source files are compiled by the compiler to the so-called **object files** usually with the suffix `.o`.
  - Object code contains relative addresses and function calls or just references to function without known implementations.
- The final executable program is created from the object files by the **linker**.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 26 / 66

Program in C Values and Variables Standard Input/Output

## Compilation and Linking Programs

- Program development is editing of the source code (files with suffixes `.c` and `.h`).
  - Human readable
  - Machine readable
- Compilation of the particular source files (`.c`) into object files (`.o` or `.obj`).
- Linking the compiled files into executable binary file.
- Execution and debugging of the application and repeated editing of the source code.

```

graph TD
    subgraph Source
        S1[Source file .c]
        S2[Header files .h]
    end
    subgraph Lib
        L[Lib files .a/.lib]
    end
    subgraph Obj
        O[Object File .o/.obj]
    end
    subgraph Executable
        E[a.out Executable binary file]
    end
    S1 --> PC[Preprocessor and Compiler]
    S2 --> PC
    PC --> O
    L --> Linker[Linker]
    O --> Linker
    Linker --> E
  
```

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 29 / 66

Program in C Values and Variables Standard Input/Output

## Escape Sequences

- Escape sequences for writing special symbols
  - `\o`, `\oo`, where `o` is an octal numeral
  - `\xh`, `\xhh`, where `h` is a hexadecimal numeral

```
1 int i = 'a';
2 int h = 0x61;
3 int o = 0141;
4 printf("i: %i h: %i o: %i c: %c\n", i, h, o, i);
5 printf("oct: \141 hex: \x61\n");
```

E.g., `\141`, `\x61` `lec01/esqdh.o.c`

- `\0` – character reserved for the end of the text string (null character)

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 24 / 66

Program in C Values and Variables Standard Input/Output

## Program Compilation and Execution

- Source file `program.c` is compiled into runnable form by the compiler, e.g., `clang` or `gcc`.
 

```
clang program.c
```
- There is a new file `a.out` that can be executed, e.g.,
 

```
./a.out
```

  - Alternatively the program can be run only by `a.out` in the case the actual working directory is set in the search path of executable files
- The program prints the argument of the function `printf()`.
 

```
./a.out
I like B3B36PRG!
```
- If you prefer to run the program just by `a.out` instead of `./a.out` you need to add your actual working directory to the search paths defined by the environment variable `PATH`.
 

```
export PATH="$PATH:pwd"
```

  - Notice, this is not recommended, because of potentially many working directories.
- The command `pwd` prints the actual working directory, see `man pwd`.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 27 / 66

Program in C Values and Variables Standard Input/Output

## Steps of Compiling and Linking

- Preprocessor** – allows to define macros and adjust compilation according to the particular environment.
  - The output is text (“source”) file.
- Compiler** – Translates source (text) file into machine readable form.
  - Native (machine) code of the platform, bytecode, or assembler alternatively.
- Linker** – links the final application from the object files.
  - Under OS, it can still reference library functions (dynamic libraries linked during the program execution), it can also contain OS calls (libraries).
- Particular steps **preprocessor**, **compiler**, and **linker** are usually implemented by a “single” program that is called with appropriate arguments.
  - E.g., `clang` or `gcc`.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 30 / 66

Program in C Values and Variables Standard Input/Output

## Compilers of C Program Language

- In PRG, we mostly use compilers from the families of compilers:
  - gcc – GNU Compiler Collection; <https://gcc.gnu.org>
  - clang – C language family frontend for LLVM. <http://clang.llvm.org>

*Under Win, two derived environments can be utilized: cygwin or MinGW.*  
<https://www.cygwin.com/> or [MinGW http://www.mingw.org/](http://www.mingw.org/)

But, it is straightforward to use WSL(2) – Windows Subsystem for Linux.

- Basic usage (flags and arguments) are identical for both compilers. *clang is compatible with gcc*

- Example
  - compile: `gcc -c main.c -o main.o`
  - link: `gcc main.o -o main`

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 31 / 66

Program in C Values and Variables Standard Input/Output

## Structure of the Source Code – Commented Example

- Commented source file `program.c`.

```

1 /* Comment is inside the markers (two characters)
2 and it can be split to multiple lines */
3 // In C99 - you can use single line comment
4 #include <stdio.h> /* The #include direct causes to include header file stdio.h from
   the C standard library */
5
6 int main(void) // simplified declaration
7 { // of the main function
8     puts("I like B3B36PRG!\n"); /* calling puts() function from the stdio.h library to
   print string to the standard output. \n denotes a new line */
9     return 0; /* termination of the function. Return value 0 to the operating system,
   0 means EXIT_SUCCESS */
10 }

```

- The program is functional; however, it is not exactly correct as `puts()` might fail. See `man puts`.

The functions `fputs()` and `puts()` return a nonnegative integer on success and EOF on error.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 32 / 66

Program in C Values and Variables Standard Input/Output

## Handling Function's Return Values

```

1 #include <stdio.h> // for puts()
2 #include <stdlib.h> // for EXIT_SUCCESS and EXIT_FAILURE
3
4 int main(void)
5 {
6     int ret = EXIT_SUCCESS;
7     if (puts("I like B3B36PRG!\n") == EOF) {
8         ret = EXIT_FAILURE;
9     }
10    return ret;
11 }

```

- An error when printing to the standard output (`stdout`) can be considered an extreme case.
- Typically, an error does not occur on standard (text) output (`stdout`), yet checking the **return values of functions is essential**, especially when working with files or loading input.
- The program is functional; however, it is not exactly correct as `puts()` might fail. Nevertheless, for the sake of clarity, in B3B36PRG, we do not require checking the return values when printing to `stdout` using functions like `puts()` or `printf()`. If such as an error occurs, it is unlikely we can somehow manage the error.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 33 / 66

Program in C Values and Variables Standard Input/Output

## Functions, Modules, and Compiling and Linking

- Function is the fundamental building block of the **modular** programming language. *Modular program is composed of several modules/source files.*
- Function definition consists of the
  - Function header;
  - Function body. *Definition is the function implementation.*
- Function prototype (declaration) is the function header to provide information how the function can be called. *It allows to use the function prior its definition, i.e., it allows to compile the code without the function implementation, which may be located in other place of the source code, or in other module.*
- Declaration is the function header and it has the form `type function_name(arguments);`

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 34 / 66

Program in C Values and Variables Standard Input/Output

## Functions in C

- Function definition inside other function is not allowed in C.
- Function names can be exported to other modules. *Module is an independent file (compiled independently).*
- Function are implicitly declared as **extern**, i.e., visible. *Local module function.*
- Using the **static** specifier, the visibility of the function can be limited to the particular module. *Local module function.*
- Function arguments are **local variables** initialized by the values passed to the function. *Arguments are passed by value (call by value).*
- C allows recursions – local variables are **automatically** allocated at the stack. *Further details about storage classes in next lectures.*
- Arguments of the function are not mandatory – void arguments. `func(void)`
- The return type of the function can be **void**, i.e., a function without return value – `void func(void);`

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 35 / 66

Program in C Values and Variables Standard Input/Output

## Program Example / Module

```

1 #include <stdio.h> /* header file */
2 #define NUMBER 5 /* symbolic constant */
3
4 int compute(int a); /* function header/prototype */
5
6 int main(int argc, char *argv[])
7 { /* main function */
8     int v = 10; /* variable definition - assignment of the memory to the variable
   name; it is also declaration that allows using the variable name from this line
   */
9     int r; /* variable definition (and declaration) */
10    r = compute(v); /* function call */
11    return 0; /* termination of the main function */
12 }
13
14 int compute(int a)
15 { /* definition of the function */
16    int b = 10 + a; /* function body */
17    return b; /* function return value */
18 }

```

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 36 / 66

Program in C Values and Variables Standard Input/Output

## Program Starting Point – main()

- Each executable program must contain a single definition of the function and that function must be the `main()`.
- The `main()` function is the starting point of the program with two basic forms.
  - Full variant for programs running under an Operating System (OS).
 

```

1 int main(int argc, char *argv[])
2 {
3     ...
4 }

```
  - For embedded systems without OS
 

```

1 int main(void)
2 {
3     ...
4 }

```

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 37 / 66

Program in C Values and Variables Standard Input/Output

## Arguments of the main() Function

- During the program execution, the OS passes to the program the number of arguments (`argc`) and the arguments (`argv`). *In the case we are using OS.*
  - The first argument is the name of the program.
 

```

1 int main(int argc, char *argv[])
2 {
3     int v;
4     v = 10;
5     v = v + 1;
6     return argc;
7 }

```

*lec01/var.c*
- The program is terminated by the `return` in the `main()` function.
- The returned value is passed back to the OS and it can be further use, e.g., to control the program execution.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 38 / 66

Program in C Values and Variables Standard Input/Output

## Example of Compilation and Program Execution

- Building the program by the `clang` compiler – it automatically joins the compilation and linking of the program to the file `a.out`. `clang var.c`
- The output file can be specified, e.g., program file `var`. `clang var.c -o var`
- Then, the program can be executed as follows. `./var`
- The compilation and execution can be joined to a single command. `clang var.c -o var; ./var`
- The execution can be conditioned to successful compilation. `clang var.c -o var && ./var`

*Programs return value — 0 means OK.*  
*Logical operator && depends on the command interpret, e.g., sh, bash, zsh.*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 39 / 66

Program in C Values and Variables Standard Input/Output

### Example – Program Execution under Shell

- The return value of the program is stored in the variable `$?`.
- Example of the program execution with different number of arguments.

```
./var
./var; echo $?
1
./var 1 2 3; echo $?
4
./var a; echo $?
2
```

*sh, bash, zsh*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 40 / 66

Program in C Values and Variables Standard Input/Output

### Writing Values of the Numeric Data Types – Literals

- Values of the data types are called **literals**
- C has 6 type of constants (literals)
  - Integer
    - Rational
  - Characters
    - Text strings
    - Enumerated
  - Symbolic – `#define NUMBER 10`

*We cannot simply write irrational numbers.*

*Enum*

*Preprocessor*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 42 / 66

Program in C Values and Variables Standard Input/Output

### Integer Literals

- Integer values are stored as one of the integer type (keywords): `int`, `long`, `short`, `char` and their **signed** and **unsigned** variants.
- Further integer data types are possible.
- Integer values (literals)
  - Decimal 123 450932
  - Hexadecimal 0x12 0xFAFF (starts with 0x or 0X)
  - Octal 0123 0567 (starts with 0)
  - unsigned 12345U (suffix U or u)
  - long 12345L (suffix L or l)
  - unsigned long 12345ul (suffix UL or ul)
  - long long 12345LL (suffix LL or ll)
- Without suffix, the literal is of the type `int`.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 43 / 66

Program in C Values and Variables Standard Input/Output

### Literals of Rational Numbers

- Rational numbers can be written
  - with floating point – `13.1`;
  - or with mantissa and exponent – `31.4e-3` or `31.4E-3`.
- Floating point numeric types depends on the implementation, but they usually follow IEEE-754-1985.
- Data types of the rational literals:
  - `double` – by default, if not explicitly specified to be another type;
  - `float` – suffix `F` or `f`;
  - `long double` – suffix `L` or `l`.

```
float f = 10.f;
long double ld = 10.11;
```

*Scientific notation*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 44 / 66

Program in C Values and Variables Standard Input/Output

### Character Literals

- Format – single (or multiple) character in apostrophe.
  - `'A'`, `'B'` or `'\n'`
- Value of the single character literal is the code of the character.
  - `'0' ~ 48`, `'A' ~ 65`
  - Value of character out of ASCII (greater than 127) depends on the compiler.*
- Type of the character constant (literal).
  - Character constant is the `int` type.**

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 45 / 66

Program in C Values and Variables Standard Input/Output

### String Literals

- Format – a sequence of character and control characters (escape sequences) enclosed in quotation (citation) marks.
- "This is a string constant with the end of line character `\n`".
  - String constants separated by white spaces are joined to single constant, e.g., `"String literal" "with the end of the line character\n"` is concatenate into `"String literal with end of the line character\n"`
- Type
  - String literal is stored in the array of the type `char` terminated by the `null` character `'\0'`.
  - E.g., String literal "word" is stored as

```
'w' 'o' 'r' 'd' '\0'
```

*The size of the array must be about 1 item longer to store \0!*  
*More about text strings in the following lectures and labs.*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 46 / 66

Program in C Values and Variables Standard Input/Output

### Constants of the Enumerated Type

- The default values of the enumeration type constants start from 0 and the values can be explicitly prescribed.
- Without specifying a value, the next element has a value one higher than the previous one.
- Enumeration type constants can have the same values.

```
1 enum {
2 SPADES,
3 CLUBS,
4 HEARTS,
5 DIAMONDS
6 };
1 enum {
2 SPADES = 10,
3 CLUBS, // the value is 11
4 HEARTS = 15,
5 DIAMONDS = 13
6 };
1 enum {
2 NUMBER_1 = 1,
3 NUMBER_2 = 2,
4 NUMBER_3 = 1,
5 NUMBER_4, // the value is 2
6 };
```

*The enumeration values are usually written in uppercase.*

- Type – enumerated constant is the `int` type.
  - Value of the enumerated literal can be used in loops.

```
1 enum { SPADES = 0, CLUBS, HEARTS, DIAMONDS, NUM_COLORS };
3 for (int i = SPADES; i < NUM_COLORS; ++i) {
4     ...
5 }
```

*lec01/enum.c*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 47 / 66

Program in C Values and Variables Standard Input/Output

### Symbolic Constant – `#define`

- Format – the constant is established by the preprocessor command `#define`.
  - It is macro command without argument.
  - Each `#define` must be on a new line.
- Symbolic constants can express constant expressions.
  - `#define MAX_1 ((10*6) - 3)`
- Symbolic constants can be nested.
  - `#define MAX_2 (MAX_1 + 1)`
- Preprocessor performs the text replacement of the define constant by its value.**
  - `#define MAX_2 (MAX_1 + 1)`

*Usually written in uppercase.*

*It is highly recommended to use brackets to ensure correct evaluation of the expression, e.g., the symbolic constant 5\*MAX\_1 with the outer brackets is 5\*((10\*6) - 3)=285 vs 5\*(10\*6) - 3=297.*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 48 / 66

Program in C Values and Variables Standard Input/Output

### Variable with a constant value modifier (keyword) (`const`)

- Using the keyword `const`, a variable can be marked as constant.
  - Compiler checks assignment and do not allow to set a new value to the variable.
- A constant value can be defined as follows.
  - `const float pi = 3.14159265;`
- In contrast to the symbolic constant.
  - `#define PI 3.14159265`
- Constant values have type, and thus it supports **type checking**.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 49 / 66

Program in C Values and Variables Standard Input/Output

### Example: Sum of Two Values

```

1 #include <stdio.h>
2 int main(void)
3 {
4     int sum; // definition of local variable of the int type
5     sum = 100 + 43; /* set value of the expression to sum */
6     printf("The sum of 100 and 43 is %i\n", sum);
7     /* %i formatting command to print integer number */
8     return 0;
9 }

```

- The variable `sum` of the type `int` represents an integer number. Its value is stored in the memory.
- `sum` is selected symbolic name of the memory location, where the integer value (type `int`) is stored.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 50 / 66

Program in C Values and Variables Standard Input/Output

### Example of Sum of Two Variables

```

1 #include <stdio.h>
2 int main(void)
3 {
4     int var1;
5     int var2 = 10; /* initialization of the variable */
6     int sum;
7     var1 = 13;
8     sum = var1 + var2;
9     printf("The sum of %i and %i is %i\n", var1, var2, sum);
10    return 0;
11 }

```

- Variables `var1`, `var2` and `sum` represent three different locations in the memory (allocated automatically), where three integer values are stored.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 51 / 66

Program in C Values and Variables Standard Input/Output

### Variable Definition

- The variable definition has a general form  
`declaration-specifiers variable-identifier;`
- Declaration specifiers are following.
  - Storage classes:** at most one of the `auto`, `static`, `extern`, `register`;
  - Type qualifiers:** `const`, `volatile`, `restrict`;

*None or more type qualifiers are allowed.*
- Type specifiers:** `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, `unsigned`.  
In addition, `struct` and `union` type specifiers can be used. Finally, own types defined by `typedef` can be used as well.

*Detailed description in further lectures.*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 52 / 66

Program in C Values and Variables Standard Input/Output

### Assignment, Variables, and Memory – Visualization unsigned char

```

1 unsigned char var1;
2 unsigned char var2;
3 unsigned char sum;
4 var1 = 13;
5 var2 = 10;
6 sum = var1 + var2;

```

- Each variable allocate 1 byte
- Content of the memory is not defined after allocation
- Name of the variable "references" to the particular memory location
- Value of the variable is the content of the memory location

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 53 / 66

Program in C Values and Variables Standard Input/Output

### Assignment, Variables, and Memory – Visualization int

```

1 int var1;
2 int var2;
3 int sum;
4 // 00 00 00 13
5 var1 = 13;
6 // x00 x00 x01 xF4
7 var2 = 500;
8 sum = var1 + var2;

```

- Variables of the `int` types allocate 4 bytes.  
*Size can be find out by the operator `sizeof(int)`.*
- Memory content is not defined after the definition of the variable to the memory.

For Intel x86 and x86-64 architectures, the values (of multi-byte types) are stored in the **little-endian order**.

500 (dec) is 0x01F4 (hex)  
513 (dec) is 0x0201 (hex)

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 54 / 66

Program in C Values and Variables Standard Input/Output

### Standard Input and Output

- An executed program within Operating System (OS) environments has assigned (usually text-oriented) standard input (`stdin`) and output (`stdout`).  
*Programs for MCU without OS does not have them.*
- The `stdin` and `stdout` streams can be utilized for communication with a user.
- Basic function for text-based input is `getchar()` and for the output `putchar()`.  
*Both are defined in the standard C library <stdio.h>.*
- For parsing numeric values the `scanf()` function can be utilized.
- The function `printf()` provides formatted output, e.g., a number of decimal places.  
*They are library functions, not keywords of the C language.*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 56 / 66

Program in C Values and Variables Standard Input/Output

### Formatted Output – printf()

- Numeric values can be printed to the standard output using `printf()`.  
*man printf or man 3 printf*
- The first argument is the format string that defines how the values are printed.
- The conversion specification starts with the character `'%'`.
- Text string not starting with `%` is printed as it is.
- Basic format strings to print values of particular types are as follows.

<code>char</code>	<code>%c</code>
<code>_Bool</code>	<code>%i, %u</code>
<code>int</code>	<code>%i, %x, %o</code>
<code>float</code>	<code>%f, %e, %g, %a</code>
<code>double</code>	<code>%f, %e, %g, %a</code>

- Specification of the number of digits is possible, as well as an alignment to left (right), etc.  
*Further options in homeworks and lab exercises.*

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 57 / 66

Program in C Values and Variables Standard Input/Output

### Formatted Input – scanf()

- Numeric values can be read (from `stdin`) by the `scanf()` function.  
*man scanf or man 3 scanf*
- The argument of the function is a format string.  
*Syntax is similar to printf().*
- A memory address of the variable has to be provided to set its value from the `stdin`.
- The return value of the `scanf()` call is the number of successfully parsed values.
- Example of readings integer value and value of the `double` type.

```

1 #include <stdio.h> // printf and scanf
2 #include <stdlib.h> // EXIT_FAILURE and EXIT_SUCCESS
3 int main(void)
4 {
5     int ret = EXIT_FAILURE;
6     int i;
7     double d;
8     printf("Enter int value: ");
9     int r = scanf("%i", &i); // operator & returns the address of i
10    if (r == 1)
11        printf("Enter a double value: ");
12    if (r == 1 && scanf("%lf", &d) == 1) { // !!! Return value !!!
13        printf("You entered %02i and %0.1f\n", i, d);
14        ret = EXIT_SUCCESS; // zero - exit success
15    }
16    return ret; // indicate failure or success
17 }

```

lec01/scanf.c

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 58 / 66

Program in C Values and Variables Standard Input/Output

### Example: Program with Output to the stdout 1/2

- Instead of `printf()` we can use `fprintf()` with explicit output stream `stdout`, or alternatively `stderr`; both functions from the `<stdio.h>`.

```

1 #include <stdio.h>
2 int main(int argc, char **argv) {
3     int r = fprintf(stdout, "My first program in C!\n");
4     fprintf(stdout, "printf() returns %d that is a number of printed characters\n", r);
5     r = fprintf(stdout, "123\n");
6     fprintf(stdout, "printf(\"123\n\") returns %d because of end-of-line '\\n\\n', r);
7     fprintf(stdout, "Its name is \"%s\\n\\n\", argv[0]);
8     fprintf(stdout, "Run with %d arguments\n", argc);
9     if (argc > 1) {
10        fprintf(stdout, "The arguments are:\n");
11        for (int i = 1; i < argc; ++i) {
12            fprintf(stdout, "Arg: %d is \"%s\\n\\n\", i, argv[i]);
13        }
14    }
15    return 0;
16 }

```

lec01/pring\_arg.c

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 59 / 66

Program in C Values and Variables Standard Input/Output

### Example: Program with Output to the stdout 2/2

- Notice, using the header file `<stdio.h>`, several other files are included as well to define types and functions for input and output. *Check by, e.g., clang -E print\_args.c*

```
./print_args first second
My first program in C!
printf() returns 23 that is a number of printed characters
123
printf("123\n") returns 4 because of end-of-line '\n'
Its name is "./print_args"
Run with 3 arguments
The arguments are:
Arg: 1 is "first"
Arg: 2 is "second"
```

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 60 / 66

Program in C Values and Variables Standard Input/Output

### Programming - Loops - Example Printed Text Message

- For example, a message can be printed 4x by repeating the print command.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     printf("I like B3B36PRG!\n");
5     printf("I like B3B36PRG!\n");
6     printf("I like B3B36PRG!\n");
7     printf("I like B3B36PRG!\n");
8     return 0;
9 }

1 #include <stdio.h>
2 int main(void)
3 {
4     const int N = 4;
5     for (int i = 0; i < N; ++i) {
6         printf("I like B3B36PRG!\n");
7     }
8     return 0;
9 }
```

- Using a loop and a control variable is the **programming approach**.
- We can generalize the example by having the user specify the number of repetitions from the standard input.

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 61 / 66

Program in C Values and Variables Standard Input/Output

### Programming - Loops – Example 1/3

```
1 #include <stdio.h>
2 #include <stdlib.h> // for EXIT_SUCCESS and EXIT_FAILURE
3 int main(void)
4 {
5     int ret = EXIT_SUCCESS;
6     int n; // variable is not initialize
7     printf("Enter a positive integer number from 1 to 9: ");
8     int r = scanf("%d", &n); // test scan has been successful in reading integer value
9     if (r == 1 && n > 0 && n < 10) {
10        int i = 0;
11        while (i < n) {
12            puts("I like B3B36PRG!");
13            i = i + 1;
14        }
15    }
16    else {
17        printf("ERROR: Input value must be in the range (0,10)\n");
18        ret = EXIT_FAILURE;
19    }
20    return ret;
21 }
```

- Naive, functional solution, in principle sufficient, but we can decompose such a program.

lec01/print.c

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 62 / 66

Program in C Values and Variables Standard Input/Output

### Programming - Loops – Example 2/3

```
1 #include <stdio.h>
2 #include <stdlib.h> // for EXIT_SUCCESS and EXIT_FAILURE
3 void print(int n);
4 int main(void)
5 {
6     int ret = EXIT_SUCCESS;
7     int n;
8     printf("Enter a positive integer number from 1 to 9: ");
9     int r = scanf("%d", &n); // passing address of the n variable
10    if (r == 1 && n > 0 && n < 10) {
11        print(n);
12    }
13    else {
14        fprintf("ERROR: Input value must be in the range (0,10)\n");
15        ret = EXIT_FAILURE;
16    }
17 }
18 return ret;
19 }
```

- Print in a separate function `print()`.
- Better, but still relatively complex – we can separate the loading, but also generalize the values and avoid „magic numbers“ in the function definition.

lec01/print2.c

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 63 / 66

Program in C Values and Variables Standard Input/Output

### Programming - Loops – Example 3/3

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int read(int min, int max, int *n)
4 {
5     printf("Enter a positive integer number
6     from %d to %d:", min, max);
7     return scanf("%d", n) == 1 && *n >= min &&
8     *n <= max; // logical true is a value !=
9     0, shortcut evaluation
10 }
11 int main(void)
12 {
13     int ret = EXIT_SUCCESS;
14     int n;
15     void print(int n)
16     {
17         int i = 0;
18         while (i < n) {
19             puts("I like B3B36PRG!");
20             i = i + 1;
21         }
22     }
23     if (read(MIN, MAX, &n)) {
24         print(n);
25     }
26     else {
27         ret = EXIT_FAILURE;
28     }
29     ret && printf("ERROR: Input value must
30     be in the range (%d,%d)\n", MIN - 1,
31     MAX + 1);
32     return ret;
33 }
```

- We pass the `read()` function a pointer to a valid memory address, it is done programmatically.
- The program returns a return value and warns the user on incorrect input. *We can also use `fprintf(stderr, ...)`.*
- We can further extend the values of `MIN` and `MAX` to be defined at compile time (`#ifndef`).

lec01/print3.c

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 64 / 66

Topics Discussed

## Summary of the Lecture

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 65 / 66

Topics Discussed

- Information about the Course
- Introduction to C Programming
  - Program, source codes and compilation of the program
  - Structure of the source code and writing program
  - Variables and basic types
  - Variables, assignment, and memory
  - Basic Expressions
  - Standard input and output of the program
  - Formating input and output
- Next: Expressions and Bitwise Operations, Selection Statements and Loops

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 66 / 66

Programs

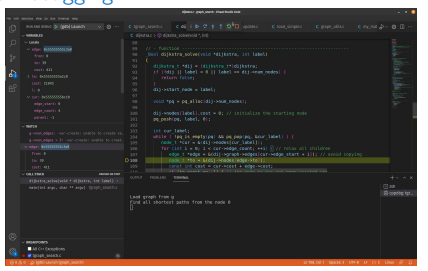
## Part IV

### Appendix

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 67 / 66

Programs

### Example of Step Debugging



<https://youtu.be/-QXu3Wk-WxQ> (~ 25 min)

**Debugging in Visual Studio Code** <https://cw.fel.cvut.cz/wiki/courses/b3b36prg/resources/coding>

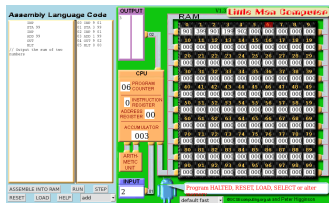
Or a two-part tutorial motivated by BAB36PRGA: <https://youtu.be/u8zrJoY1NU> (~11 minutes) and <https://youtu.be/g2904yHTAdQ> (~15 minutes).

Jan Faigl, 2025 B3B36PRG – Lecture 01: Introduction to C Programming 68 / 66

## Computer Calculation

- Understanding of the calculation on a processor simulator such as Little Man Computer.
  - <https://peterhigginson.co.uk/LMC/>, <https://gcsecomputing.org.uk/lmc/>
  - <http://www.vivaxsolutions.com/web/lmc.aspx>, <https://www.youtube.com/watch?v=6cbJW4AGmk>

- LDA** – Load to the acc.
- STA** – Store the acc. to address
- ADD** – Add to the acc.
- INP** – Input to the acc.
- OUT** – Output of the acc.
- BRP** – Set PC on zero or positive acc.
- HLT** – Stop executing program



## Example – Processing the Source Code by Preprocessor

- Using the **-E** flag, we can perform only the preprocessor step.
 

```
gcc -E var.c
```

Alternatively `clang -E var.c`

```
1 # 1 "var.c"
2 # 1 "<built-in>"
3 # 1 "<command-line>"
4 # 1 "var.c"
5 int main(int argc, char **argv) {
6     int v;
7     v = 10;
8     v = v + 1;
9     return argc;
10 }
```

lec01/var.c

## Example – Compilation of the Source Code to Assembler

- Using the **-S** flag, the source code can be compiled to Assembler.
 

```
clang -S var.c -o var.s
```

```
1 .file "var.c"
2 .text
3 .globl main
4 .align 16, 0x90
5 .type main,@function
6 main:
7     @main
8     .cfi_startproc
9     # BB#0:
10    pushq %rbp
11    .Ltmp2:
12    .cfi_def_cfa_offset 16
13    .cfi_offset %rbp, -16
14    movq %rsp, %rbp
15    .Ltmp4:
16    .cfi_def_cfa_register %rbp
17    movl $0, -4(%rbp)
18    movl %edi, -8(%rbp)
19    movq %rsi, -16(%rbp)
20    movl $10, -20(%rbp), %edi
21    addl $1, %edi
22    movl %edi, -20(%rbp)
23    popq %rbp
24    ret
25
26 .Ltmp5:
27 .size main, .Ltmp5-main
28 .cfi_endproc
29
30 .ident "FreeBSD clang version 3.4.1 (tags
31 /RELEASE_34/dot1-final 208032) 20140512"
32 .section ".note.GNU-stack","",@progbits
```

## Example – Compilation to Object File

- The source file is compiled to the object file.
 

```
clang -c var.c -o var.o
```

`clang -c var.c -o var.o`

```
% clang -c var.c -o var.o
% file var.o
var.o: ELF 64-bit LSB relocatable, x86-64, version 1 (FreeBSD), not stripped
```

- Linking** the object file(s) provides the executable file.
 

```
clang var.o -o var
```

`clang var.o -o var`

```
% clang var.o -o var
% file var
var: ELF 64-bit LSB executable, x86-64, version 1 (FreeBSD), dynamically linked (uses shared
libs), for FreeBSD 10.1 (1001504), not stripped
```

dynamically linked  
not stripped

## Example – Executable File under OS 1/2

- By default, executable files are “tied” to the C library and OS services.
- The dependencies can be shown by `ldd var`.
 

```
ldd var
var:
    libc.so.7 => /lib/libc.so.7 (0x2c41d000)
```

`ldd - list dynamic object dependencies`

- The so-called static linking can be enabled by the **-static**.
 

```
clang -static var.o -o var
% ldd var
% file var
var: ELF 64-bit LSB executable, x86-64, version 1 (FreeBSD), statically linked, for FreeBSD
10.1 (1001504), not stripped
% ldd var
ldd: var: not a dynamic ELF executable
```

Check the size of the created binary files!

## Example – Executable File under OS 2/2

- The compiled program (object file) contains symbolic names (by default).
 

```
clang var.c -o var
wc -c var
7240 var
```

*E.g., usable for debugging.*

`wc - word, line, character, and byte count`  
`-c - byte count`

- Symbols can be removed by the tool (program) **strip**.
 

```
strip var
wc -c var
4888 var
```

Alternatively, you can show size of the file by the command `ls -l`.

## Extended Variants of the main() Function

- Extended declaration of the `main()` function provides access to the environment variables.
 

*For Unix and MS Windows like OS.*

```
1 int main(int argc, char **argv, char **envp) { ... }
```

The environment variables can be accessed using the function `getenv()` from the standard library `<stdlib.h>`.

lec01/main\_env.c

- For Mac OS X, there are further arguments.

```
1 int main(int argc, char **argv, char **envp, char **apple)
2 {
3     ...
4 }
```