Introduction to C Programming

Jan Faigl

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

Lecture 01

PRG – Programming in C

Overview of the Lecture

- Part 1 – Course Organization
  - Course Goals
  - Means of Achieving the Course Goals
  - Evaluation and Exam
- Part 2 – Introduction to C Programming
  - Programs
  - 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 homeworks – **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 Goals

- Master (yourself) programming skills.

  Labs, homeworks, 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`

Course Organization and Evaluation

- 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 **180 hours per semester**
  - Contact part (lecture and labs): 3 hours per week, i.e., 42 hours in the total
  - Exam including preparation: *10 hours*
  - Home preparation (first **book reading** and followed by homeworks) 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 homeworks, and semestral project.

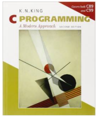
- **Consultation** - If you do not know, or spent too much time with the homework, consult with the instructor/lecturer.
  - **Maximize the contact time during labs and lectures, ask questions, and discuss.**

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 deadline is in 18.3.2023.

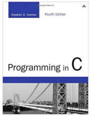
- Lectures – support for the textbook, slides, comments, and **your notes**.

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

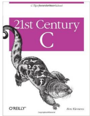
- Laboratory exercises – gain practical skills by doing homeworks (yourself).

Further Books

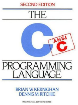
- Programming in C, 4th Edition, *Stephen G. Kochan*, Addison-Wesley, 2014, ISBN 978-0321776419



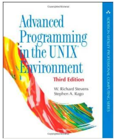
- 21st Century C: C Tips from the New School, *Ben Klemens*, O'Reilly Media, 2012, ISBN 978-1449327149



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

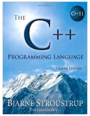


- Advanced Programming in the UNIX Environment, 3rd edition, *W. Richard Stevens, Stephen A. Rago* Addison-Wesley, 2013, ISBN 978-0-321-63773-4

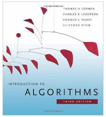


Further Resources

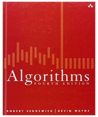
- The C++ Programming Language, 4th Edition (C++11) , *Bjarne Stroustrup*, Addison-Wesley, 2013, ISBN 978-0321563842



- 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



Lectures – Spring Semester Academic Year 2024/2024

- Schedule for the academic year 2023/2024.

  https://intranet.fel.cvut.cz/cz/education/harmonogram.html

- Lectures:
  - Dejvice, Lecture Hall No. T2:D3-209, Tuesday, 16:15-17:45.
- 14 teaching weeks - (19.2.–26.5.2024); 13 weeks in practice.
  - National holiday – 01.04.2024 (Monday).
  - National holiday – 01.05.2024 (Wednesday).
  - National holiday – 08.05.2024 (Wednesday).
  - Rector's day – 14.05.2023 (Tuesday).
  - Thursday 09.05.2024 – classes as on Wednesday (odd teaching week).

Teachers

- RNDr. **Ingrid Nagyová**, Ph.D.

- MSc. **Yuliia Prokop**, Ph.D.

- Ing. **Martin Zoula**

Communicating Any Issues Related to the Course

- Ask the lab teacher or the lecturer.
- Use e-mail for communication.
  - Use your faculty e-mail.
  - Put PRG or B3B36PRG to the subject of your message.
  - Send copy (Cc) to lecturer/teacher.
- *Discord channel*.

Computers and Development Tools

- Computer labs - network boot. Sync your files using, e.g., ownCloud, gdrive, ssh, ftp.
  - **You have to set your password via `https://felk.cvut.cz` – rooms of Dept. of Computer Science.**
  - **You need the access for implementation exam.**
- Compilers **gcc** or **clang**. `https://gcc.gnu.org` or `http://clang.llvm.org`
- Project building **make** (GNU make). *Examples of usage on lectures and labs.*
- Text editor – `gedit`, **atom**, **sublime**, **vim**. `https://atom.io/`, `http://www.sublimetext.com/` `http://www.root.cz/clanky/textovy-editor-vim-jako-ide`
- **Visual Studio Code** – `code` – great for editing and terminal based compilation.
- C/C++ development environments – **WARNING: Do Not Use An IDE** at the beginning, to become familiar with the syntax. `http://c.learncodethehardway.org/book/ex0.html`
  - **Visual Studio Code**; CLion – `https://www.jetbrains.com/clion`; `Code::Blocks`, `CodeLite`, `NetBeans (C/C++)`, `Eclipse-CDT`.
- **Embedded development for the Nucleo**.
  - `ARMmbed` – `https://os.mbed.com/platforms/ST-Nucleo-F446RE/`
  - `https://studio.keil.arm.com/`
  - `System Workbench for STM32` (based on Eclipse); `direct cross-compiling` using makefiles.

Services – Academic Network, FEE, CTU

- http://www.fel.cvut.cz/cz/user-info/index.html
- Cloud storage ownCloud – https://owncloud.cesnet.cz
- Sending large files – https://filesender.cesnet.cz
- Schedule, deadlines – FEL Portal, https://portal.fel.cvut.cz
- **FEL Google Account** – access to Google Apps for Education

  See http://google-apps.fel.cvut.cz/
- Gitlab FEL – https://gitlab.fel.cvut.cz/
- Information resources (IEEE Xplore, ACM, Science Direct, Springer Link)

  https://dialog.cvut.cz
- Academic and campus software license

  https://download.cvut.cz
- National Super Computing Grid Infrastructure – MetaCentrum

  http://www.metacentrum.cz/cs/index.html

Homeworks

- 1+7 homeworks - seven for the workstation.

https://cw.fel.cvut.cz/wiki/courses/b3b36prg/hw/start

1. HW 00 – **Testing** (1 point) *1 h*
2. HW 01 – **ASCII Art** (2 points) *3 h*

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

Late submission is penalized!

- Coding style needs to be learn, 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.

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 task can be awarded up to **20 points**.
- Bonus part can be awarded for additional **10 points**.

  Up to 30 points in the total for the semestral project.

- Minimum required points: 10!

Deadline – best before 17.05.2024.

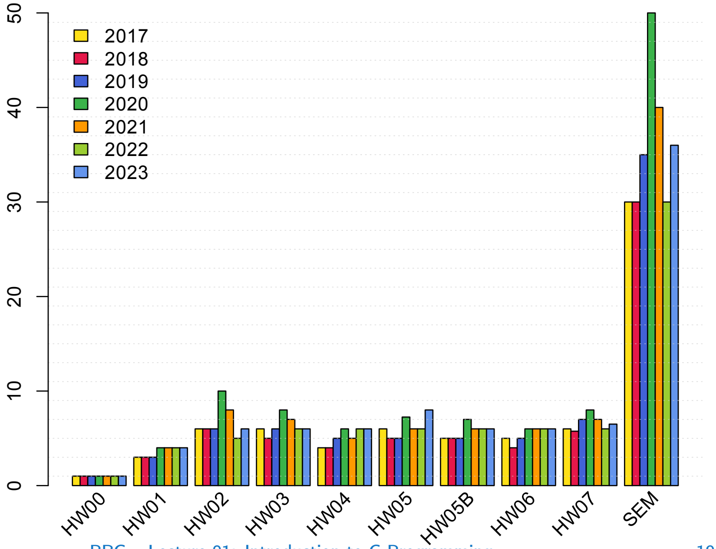
Further updates and additional points might be possible!

Deadline – 19.05.2024.

- Expected required time to finish the semestral project is about 30–50 hours.

Expected Time Needed to Complete Homeworks

- B3B36PRG - Average sum of the reported median times.
  - **96 hours** (with HW05B $\sim$ 6 h, SEM $\sim$ 30 h).
- *6 credits is about 150–180 hours that is*
  - *42 h contact part*
  - *10 h exam, and*
  - *about 100–128 hours for homeworks.*
- **Plan your work! Use the first weeks to read the textbook!**



Course Evaluation

Points	Maximum Points	Required Points	Minimum Points
Homeworks and labs	40	25	
Semester project	30	10	
Exam test	20		**10**
Implementation exam	20		10
Total	110 points	35 points is F!	

- **25** points from the homeworks and 10 points from the semestral project are required for awarding ungraded assessment.
- The course can be passed with **ungraded assessment** and **exam**.
- All homeworks must be submitted and they have to pass the mandatory assessment.

Overview of the Lectures

1. Course information, Introduction to C programming *K. N. King: chapters 1, 2, and 3*
2. Writing your program in C, control structures (loops), expressions *K. N. King: chapters 4, 5, 6, and 20*
3. Data types, arrays, pointer, memory storage classes, function call *K. N. King: chapters 7, 8, 9, 10, 11, and 18*
4. Data types: arrays, strings, and pointers *K. N. King: chapters 8, 11, 12, 13, and 17*
5. Data types: Struct, Union, Enum, Bit fields. Preprocessor and Large Programs *K. N. King: chapters 10, 14, 15, 16, and 20*
6. Input/Output – reading/writting from/to files and other communication channels, Standard C library – selected functions *K. N. King: chapters 21, 22, 23, 24, 26, and 27*
7. Parallel and multi-thread programming – methods and synchronizations primitives
8. Multi-thread application models, POSIX threads and C11 threads
9. C programming language wrap up, examples such as linked lists
10. *ANSI C, C99, C11 and differences between C and C++* Introduction to C++.
11. Quick introduction to C++
12. C++ Constructions ion coding examples<F10>

    Reserve *(Rector's day)*
13. *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!

Part II

Part 2 – Introduction to C Programming

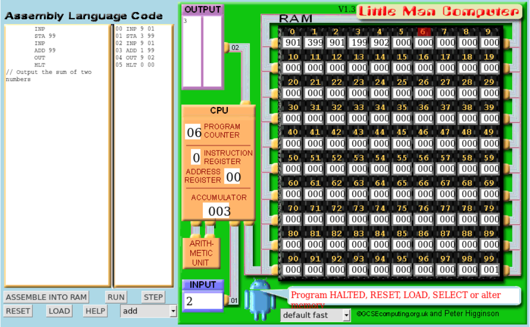
Computer Calculation

- Understanding of the calculation on a procesor 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=6cbJWV4AGmk

- **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 possitive acc.
- **HLT** – Stop executing program



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 became familiar with the memory model and key elements for writting 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.

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).

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

```
int i = 'a';
int h = 0x61;
int o = 0141;

printf("i: %i h: %i o: %i c: %c\n", i, h, o, i);
printf("oct: \141 hex: \x61\n");
```

E.g., \141, \x61 lec01/esqdho.c

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

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 a _.
  - 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$_{32}$

  auto break case char const continue default do double else enum extern float for 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`.

Simple C Program

```c
#include <stdio.h>

int main(void)
{
   printf("I like B3B36PRG!\n");

   return 0;
}
```

lec01/program.c

- 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.

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`.

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.

1. 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.

2. 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

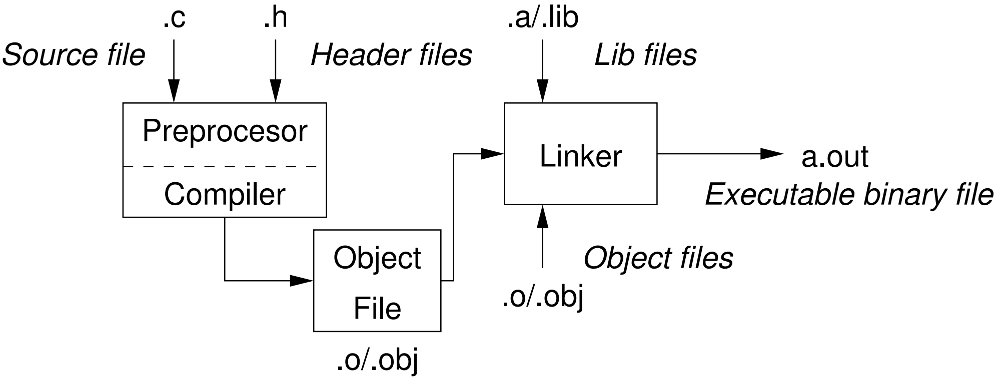
   The command combines preprocessor and compiler.

3. Executable file is linked from the particular object files and referenced libraries by the linker (linking), e.g.,

   clang program.o -o program

Compilation and Linking Programs

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



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`*.*

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*** https://www.cygwin.com/ *or* ***MinGW*** http://www.mingw.org/

- 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`

Structure of the Source Code – Commented Example

- Commented source file program.c.

```
/* Comment is inside the markers (two characters)
   and it can be split to multiple lines */
// In C99 - you can use single line comment
#include <stdio.h> /* The #include direct causes to include header file
    stdio.h from the C standard library */

int main(void) // simplified declaration
{              // of the main function
   printf("I like B3B36PRG!\n"); /* calling printf() function from the
    stdio.h library to print string to the standard output. \n denotes
    a new line */
   return 0; /* termination of the function. Return value 0 to the
    operating system */
}
```

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);
```

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.
- 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.

  `fnc(void)`
- The return type of the function can be `void`, i.e., a function without return value – `void fnc(void);`

Program Example / Module

```c
#include <stdio.h> /* header file */
#define NUMBER 5 /* symbolic constant */

int compute(int a); /* function header/prototype */

int main(int argc, char *argv[])
{ /* main function */
   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 */
   int r; /* variable definition (and declaration) */
   r = compute(v); /* function call */
   return 0; /* termination of the main function */
}

int compute(int a)
{ /* definition of the function */
  int b = 10 + a; /* function body */
  return b; /* function return value */
}
```

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.
  1. Full variant for programs running under an Operating System (OS).

     ```
     int main(int argc, char *argv[])
     {
         ...
     }
     ```

  2. For embedded systems without OS

     ```
     int main(void)
     {
         ...
     }
     ```

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.

```c
int main(int argc, char *argv[])
{
   int v;
   v = 10;
   v = v + 1;
   return argc;
}
```

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.

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`.

Example – Program Execution under Shell

- The return value of the program is stored in the variable **$?**.

  sh, bash, zsh

- Example of the program execution with different number of arguments.

```
./var

./var; echo $?
1

./var 1 2 3; echo $?
4

./var a; echo $?
2
```

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 "var.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "var.c"
int main(int argc, char **argv) {
    int v;
    v = 10;
    v = v + 1;
    return argc;
}
```

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
```

```
    .file "var.c"
    .text
    .globl  main
    .align  16, 0x90
    .type main,@function
main:
        # @main
    .cfi_startproc
# BB#0:
    pushq %rbp
.Ltmp2:
    .cfi_def_cfa_offset 16
.Ltmp3:
    .cfi_offset %rbp, -16
    movq  %rsp, %rbp
.Ltmp4:
    .cfi_def_cfa_register %rbp
    movl  $0, -4(%rbp)
    movl  %edi, -8(%rbp)
    movq  %rsi, -16(%rbp)
    movl  $10, -20(%rbp)
    movl  -20(%rbp), %edi
    addl  $1, %edi
    movl  %edi, -20(%rbp)
    movl  -8(%rbp), %eax
    popq  %rbp
    ret
.Ltmp5:
    .size main, .Ltmp5-main
    .cfi_endproc


    .ident  "FreeBSD clang version 3.4.1 (
      tags/RELEASE_34/dot1-final 208032)
      20140512"
    .section  ".note.GNU-stack","",
      @progbits
```

Example – Compilation to Object File

- The souce file is compiled to the object file.

  `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
% 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 – list dynamic object dependencies

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

- 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).

E.g., usable for debugging.

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

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`.

Writting Values of the Numeric Data Types – Literals

- Values of the data types are called **literals**
- C has 6 type of constants (literals)
  - Integer
  - Rational

    We cannot simply write irrational numbers.
  - Characters
  - Text strings
  - Enumerated

    Enum
  - Symbolic – `#define NUMBER 10`

    Preprocessor

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 typu `int`.

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`.

Scientific notation

- Floating point numeric types depends on the implementation, but they usually follow IEEE-754-1985. `float`, `double`
- Data types of the rational literals:
  - `double` – by default, if not explicitly specified to be another type;
  - `float` – suffix `F` or `f`;

`float f = 10.f;`

  - `long double` – suffix `L` or `l`.

`long double ld = 10.1l;`

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'`$\sim$ 48, `'A'`$\sim$ 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.**

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.

Constants of the Enumerated Type

- By default, values of the enumerated type starts from 0 and each other item increase the value about one, values can be explicitly prescribed.

```
enum {
   SPADES,
   CLUBS,
   HEARTS,
   DIAMONDS
};
```

```
enum {
   SPADES = 10,
   CLUBS, /* the value is 11 */
   HEARTS = 15,
   DIAMONDS = 13
};
```

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.

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

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.

```
#define SCORE 1
```

Usually written in uppercase.

- 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)
```

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.*

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**.

Example: Sum of Two Values

```
#include <stdio.h>

int main(void)
{
  int sum; // definition of local variable of the int type

  sum = 100 + 43; /* set value of the expression to sum */
  printf("The sum of 100 and 43 is %i\n", sum);
  /* %i formatting command to print integer number */
  return 0;
}
```

- 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.

Example of Sum of Two Variables

```c
#include <stdio.h>

int main(void)
{
   int var1;
   int var2 = 10; /* inicialization of the variable */
   int sum;

   var1 = 13;

   sum = var1 + var2;

   printf("The sum of %i and %i is %i\n", var1, var2, sum);

   return 0;
}
```

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

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 quantifiers**: `const`, `volatile`, `restrict`;

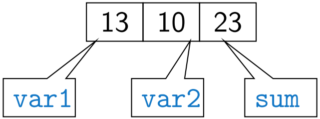
    None or more type quantifiers are allowed.
  - **Type specifiers**: `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, `unsigned`. In addition, `struct` and `union` type specifiers can be used. Finally, own types defined by `typedef` can be used as well.

Detailed description in further lectures.

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

```c
unsigned char var1;
unsigned char var2;
unsigned char sum;

var1 = 13;
var2 = 10;

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



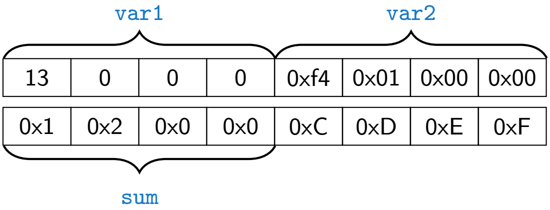
Assignment, Variables, and Memory – Visualization `int`

```c
int var1;
int var2;
int sum;

// 00 00 00 13
var1 = 13;

// x00 x00 x01 xF4
var2 = 500;

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.



500 (dec) is 0x01F4 (hex)

513 (dec) is 0x0201 (hex)

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

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.

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.

`char`	`%c`
`_Bool`	`%i`, `%u`
`int`	`%i`, `%x`, `%o`
`float`	`%f`, `%e`, `%g`, `%a`
`double`	`%f`, `%e`, `%g`, `%a`

- Specification of the number of digits is possible, as well as an alignment to left (right), etc.

  Further options in homeworks and lab exercises.

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.

```c
#include <stdio.h>

int main(void)
{
   int i;
   double d;

   printf("Enter int value: ");
   int r = scanf("%i", &i); // operator & returns the address of i
   if (r == 1)
      printf("Enter a double value: ");
   if (scanf("%lf", &d) == 1) {
      printf("You entered %02i and %0.1f\n", i, d);
   }
   return 0; // return value of main() - zero is exit success
}
```

lec01/scanf.c

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>`.

```c
#include <stdio.h>

int main(int argc, char **argv) {
   int r = fprintf(stdout, "My first program in C!\n");
   fprintf(stdout, "printf() returns %d that is a number of printed characters\n", r);
   r = fprintf(stdout, "123\n");
   fprintf(stdout, "printf(\"123\\n\") returns %d because of end-of-line '\\n'\n", r);
   fprintf(stdout, "Its name is \"%s\"\n", argv[0]);
   fprintf(stdout, "Run with %d arguments\n", argc);
   if (argc > 1) {
      fprintf(stdout, "The arguments are:\n");
      for (int i = 1; i < argc; ++i) {
         fprintf(stdout, "Arg: %d is \"%s\"\n", i, argv[i]);
      }
   }
   return 0;
}
```

lec01/pring_args.c

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"
```

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.*

```
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.

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

Summary of the Lecture

Topics Discussed

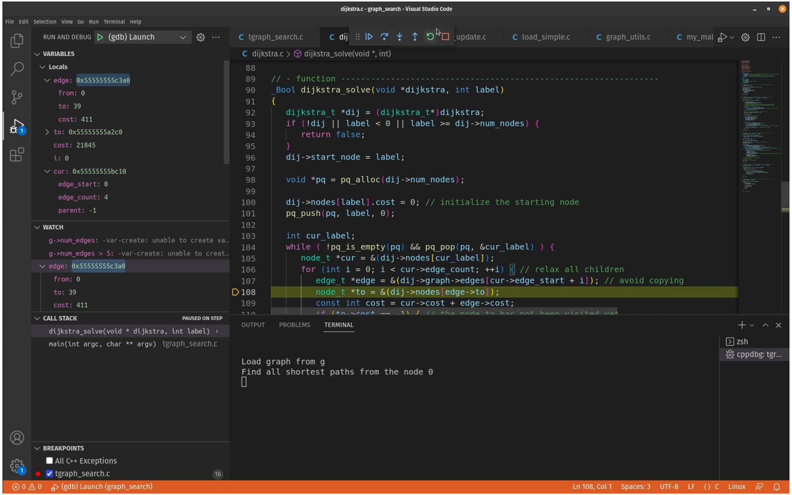
- Information about the Course
- Introduction to C Programming
  - Program, source codes and compilation of the program
  - Structure of the souce code and writting program
  - Variables and basic types
  - Variables, assignment, and memory
  - Basic Expressions
  - Standard input and output of the program
  - Formating input and output

- Next: Expressions and Bitwise Operations, Selection Statements and Loops

Part IV

Appendix

Example of step debugging



https://youtu.be/rTv_ypcm9XI (∼ 25 min)