Introduction to C Programming

Jan Faigl

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

Lecture 01

BE5B99CPL – C Programming Language

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
  - Program in C
  - Values and Variables
  - Expressions
  - Standard Input/Output

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

Part I

Part 1 – Course Organization

Course and Lecturer

BE5B99CPL – C Programming Language

- Course web page
  https://cw.fel.cvut.cz/wiki/courses/be5b99cpl
- Submission of the homework
  Individually during the labs
- Lecturer:
  - doc. 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

  - Multithreading applications;
  - Embedded applications – `Nucleo F401RE`

Course Organization and Evaluation

- BE5B99CPL– C Programming Language
- Extent of teaching: 2(lec)+2(lab)+5(hw);
- Completion: Z,ZK; Credits: 6;

Z – ungraded assessment, ZK – exam

- Ongoing work during the semester – homeworks and test
- Exam: test and implementation exam

Be able to independently work with the computer in the lab (class room)

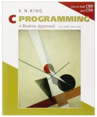
- Attendance to labs and submission of homeworks

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

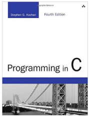
- Lectures – support for the textbook, slides, comments, and **your notes**

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

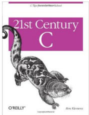
- Laboratory Exercises – gain practical skills by doing homeworks (yourself).

Further Books

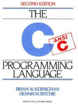
- Programming in C, 4th Edition, *Stephen G. Kochan*, Addison-Wesley, 2014, ISBN 978-0321776419



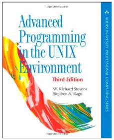
- 21st Century C: C Tips from the New School, *Ben Klemens*, O'Reilly Media, 2012, ISBN 978-1449327149



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

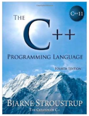


- Advanced Programming in the UNIX Environment, 3rd edition, *W. Richard Stevens, Stephen A. Rago* Addison-Wesley, 2013, ISBN 978-0-321-63773-4

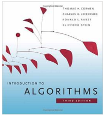


Further Resources

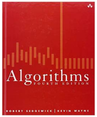
- The C++ Programming Language, 4th Edition (C++11) , *Bjarne Stroustrup*, Addison-Wesley, 2013, ISBN 978-0321563842



- 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 – Winter Semester (WS) Academic Year 2017/2018

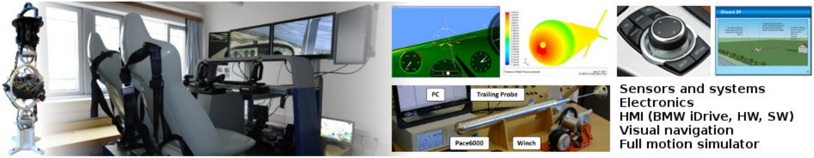
- Schedule for the academic year 2017/2018

  http://www.fel.cvut.cz/en/education/calendar.html
- Lectures:
  - Karlovo náměstí, Room No. KN:E-307, Wednesday, 9:15–10:45
- 14 teaching weeks

  13 lectures

Teachers

- doc. Ing. **Pavel Pačes**, Ph.D.
  - Department of Computer Science Measurements – http://www.pacespavel.net/
  - Center for Advanced Simulation and Technology



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 CPL or BE5B99CPL to the subject of your message
  - Send copy (Cc) to lecturer/teacher

Computers and Development Tools

- Network boot with home directories (NFS v4)
  Data transfer and file synchronizations – ownCloud, SSH, FTP, USB
- 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
- C/C++ development environments – **WARNING: Do Not Use An IDE**
  http://c.learncodethehardway.org/book/ex0.html
  - Debugging – gdb, cgdb, ddd
  - Code::Blocks, CodeLite
    http://www.codeblocks.org, http://codelite.org
  - NetBeans 8.0 (C/C++), Eclipse-CDT
  - CLion – https://www.jetbrains.com/clion
- **Embedded development for the Nucleo** – System Workbench for STM32
  based on Eclipse

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

- Six homeworks for the workstation and embedded Nucleo platform

  https://cw.fel.cvut.cz/wiki/courses/be5b99cpl/hw/start

- The final homework HW 06 combines an application running on the Nucleo board that communicates with the workstation program.

1. HW 01 (5 points) – Processing input data and computing statistics
2. HW 02 (5 points) – First program on embedded platform
3. HW 03 (10 points) – Reading/writing (files and other communications)
4. HW 04 (10 points) – Readings and visualization on the embedded platform
5. HW 05 (10 points) – Multi-thread computation
6. HW 06 (20 points) – A complex multi-threaded application with communication Workstation/Nucleo

Some adjustments are expected

- All homeworks must be submitted to award an ungraded assessment
- Late submission will be penalized!

Course Evaluation

Points	Maximum Points	Required Minimum Points
Homeworks	60	30
Test	10	**0**
Exam test	20	10
Implementation exam	10	0
Total	100 points	40 points is F!

- **30** points from the semester are required for awarding ungraded assessment
- The course can be passed with **ungraded assessment** and **exam**
- All homeworks must be submitted

Grading Scale

Grade	Points	Mark	Evaluation
A	≥ 90	1	Excellent
B	80–89	1,5	Very Good
C	70–79	2	Good
D	60–69	2,5	Satisfactory
E	50–59	3	Sufficient
F	$<$50	4	Fail

- Expected results
  - All homeworks work and have been submitted before the deadlines (60 points)
  - Test (10 points)
  - Exam (test) (20 points)
  - Exam (implementation) (10 points)
  - **95 points** and more (A – Excellent) – with small imperfection
  - **76 points** (C – Good) for 20% loss

76 and more points represents a solid background for further development of your programming skills.

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, and 6
3. Data types, arrays, pointer, memory storage classes, function call
   K. N. King: chapters 7, 8, 9, 10, and 11
4. Data types: arrays, strings, and pointers *K. N. King: chapters 11, 12, and 13*
5. Data types: Struct, Union, Enum, Bit fields. Preprocessor and Large Programs.
   K. N. King: chapters 14, 15, and 16
6. Input/Output – reading/writing from/to files and other communication channels
   - Standard C Library – selected functions and libraries
7. Parallel and multi-thread programming – methods and synchronizations primitives
8. Multi-thread application models, POSIX threads and C11 threads
9. *Version Control Systems (VCS)*
10. *ANSI C, C99, C11 – differences and extensions*
11. *Differences between C and C++: Introduction to object oriented programming in C++*
12. *Object oriented programming in C++: classes, objects, encapsulation, and polymorphism*

Part II

Part 2 – Introduction to C Programming

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 CPL 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 need to use functions from available binary libraries

Valid Characters for Writing Source Codes in C

- Lowercase and uppercase letters, numeric characters, symbols and separators **ASCII** – American Standard Code for Information Interchange
  - a–z A–Z 0—9
  - ! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] ^ _ { | } ~
  - space, tabular, new line
- Escape sequences for writting special symbols
  - \' – ', \" – ", \? – ?, \\– \
- Escape sequences for writting numeric values in a text string
  - \o, \oo, where o is an octal numeral
  - \xh, \xhh, where h is a hexadecimal numeral

```c
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`

Writing Codes in C

- Each executable program must have at least one function and the function has to be `main()`
- The run of the program starts at the beginning of the function `main()`, e.g.,

```c
#include <stdio.h>

int main(void)
{
   printf("I like BE5B99CPL!\n");

   return 0;
}
```

- The form of the `main()` function is prescribed

See further examples in this lecture

Simple C Program

```c
#include <stdio.h>

int main(void)
{
   printf("I like BE5B99CPL!\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 BE5B99CPL!
```

- 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`

Structure of the Source Code – Commented Example

- Commented source file `program.c`

```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 BE5B99CPL!\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 */
}
```

Program Building: Compiling and Linking

- The previous example combines three steps of building the program into 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

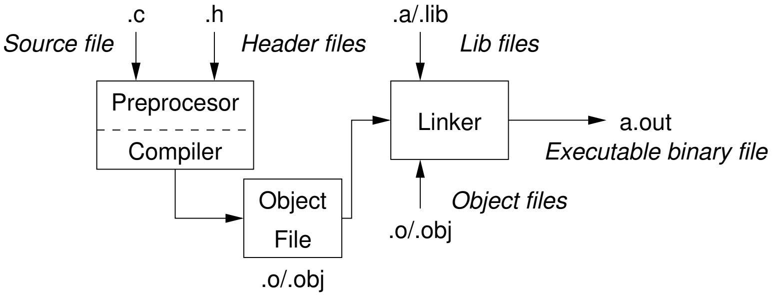
   The command combines preprocessor and compiler.

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 compilation 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 contains 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 CPL, 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`

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);`

Example of Program / Module

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

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

int main(int argc, char *argv[])
{ /* main function */
   int v = 10; /* variable declaration */
   int r;
   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 at least one definition of the function and that function must be the `main()`
- The `main()` function is the starting point of the program
- The `main()` has two basic forms
  1. Full variant for programs running under an Operating System (OS)

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

     - It can be alternatively written as

       ```
       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

```
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

  ```
  ./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` compiler option

```
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 = 10f;`
  - `long double` – suffix `L` or `l`

    `long double ld = 10l;`

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

- Format
  - 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

```c
#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 commend 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 Declaration

- The variable declaration has general form

  declaration-specifiers declarators;

- Declaration specifiers are:
  - **Storage classes**: at most one of the `auto`, `static`, `extern`, `register`
  - **Type quantifiers**: `const`, `volatile`, `restrict`

    Zero 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.

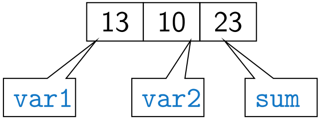
Assignment, Variables, and Memory – Visualization `unsigned char`

```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

var1				var2			
13	0	0	0	0xf4	0x01	0x00	0x00
0x1	0x2	0x0	0x0	0xC	0xD	0xE	0xF
sum							

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.*

Expressions

- **Expression** prescribes calculation value of some given input
- Expression is composed of **operands**, **operators**, and **brackets**
- Expression can be formed of
  - literals
  - variables
  - constants
  - unary and binary operators
  - function calling
  - brackets
- The order of operation evaluation is prescribed by the operator **precedence** and **associativity**.

Example

```
10 + x * y      // order of the evaluation 10 + (x * y)
10 + x + y      // order of the evaluation (10 + x) + y
```

** has higher priority than +*
+ is associative from the left-to-right

Operators

- Operators are selected characters (or a sequences of characters) dedicated for writting expressions
- Five types of binary operators can be distinguished
  - **Arithmetic** operators – additive (addition/subtraction) and multiplicative (multiplication/division)
  - **Relational** operators – comparison of values (less than, greater than, . . .)
  - **Logical** operators – logical `AND` and `OR`
  - **Bitwise** operators – bitwise `AND`, `OR`, `XOR`, bitwise shift (left, right)
  - Assignment operator $=$ – a variables (l-value) is on its left side
- Unary operators
  - Indicating positive/negative value: $+$ and $-$

    Operator $-$ modifies the sign of the expression
  - Modifying a variable : $++$ and $--$
  - Logical negation: **!**
  - Bitwise negation: $\sim$
- Ternary operator – conditional expression **? :**

Variables, Assignment Operator, and Assignment Statement

- Variables are defined by the type and name
  - Name of the variable are in lowercase
  - Multi-word names can be written with underscore _

    Or we can use CamelCase
  - Each variable is defined at new line

```
int n;
int number_of_items;
int numberOfItems;
```

- Assignment is setting the value to the variable, i.e., the value is stored at the memory location referenced by the variable name
- Assignment operator

$$\langle \textbf{l-value} \rangle = \langle \textbf{expression} \rangle$$

Expression is literal, variable, function calling, ...

  - The side is the so-called **l-value – location-value, left-value**

    It must represent a memory location where the value can be stored.
  - Assignment is an expression and we can use it everywhere it is allowed to use the expression of the particular type.
- Assignment statement is the assignment operator = and ;

Basic Arithmetic Expressions

- For an operator of the numeric types `int` and `double`, the following operators are defined

Also for `char`, `short`, and `float` numeric types.

  - Unary operator for changing the sign $-$
  - Binary addition $+$ and subtraction $-$
  - Binary multiplication `*` and division $/$
- For integer operator, there is also
  - Binary module (integer reminder) **%**
- If both operands are of the same type, the results of the arithmetic operation is the same type
- In a case of combined data types **int** and **double**, the data type `int` is converted to **double** and the results is of the **double** type.

Implicit type conversion

Example – Arithmetic Operators 1/2

```c
int a = 10;
int b = 3;
int c = 4;
int d = 5;
int result;

result = a - b; // subtraction
printf("a - b = %i\n", result);

result = a * b; // multiplication
printf("a * b = %i\n", result);

result = a / b; // integer divison
printf("a / b = %i\n", result);

result = a + b * c; // priority of the operators
printf("a + b * c = %i\n", result);

printf("a * b + c * d = %i\n", a * b + c * d);          // -> 50
printf("(a * b) + (c * d) = %i\n", (a * b) + (c * d)); // -> 50
printf("a * (b + c) * d = %i\n", a * (b + c) * d);      // -> 350
```

lec01/arithmetic_operators.c

Example – Arithmetic Operators 2/2

```c
#include <stdio.h>

int main(void)
{
   int x1 = 1;
   double y1 = 2.2357;
   float x2 = 2.5343f;
   double y2 = 2;

   printf("P1 = (%i, %f)\n", x1, y1);
   printf("P1 = (%i, %i)\n", x1, (int)y1);
   printf("P1 = (%f, %f)\n", (double)x1, (double)y1);
   printf("P1 = (%.3f, %.3f)\n", (double)x1, (double)y1);

   printf("P2 = (%f, %f)\n", x2, y2);

   double dx = (x1 - x2); // implicit data conversion to float
   double dy = (y1 - y2); // and finally to double

   printf("(P1 - P2)=(%.3f, %0.3f)\n", dx, dy);
   printf("|P1 - P2|^2=%.2f\n", dx * dx + dy * dy);
   return 0;
}
```

lec01/points.c

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

`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 from the standard input can be read using the `scanf()` function *`man scanf` or `man 3 scanf`*
- The argument of the function is a format string
  Syntax is similar to `printf()`
- It is necessary to provide a memory address of the variable to set its value from the `stdin`
- 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: ");
   scanf("%i", &i); // operator & returns the address of i

   printf("Enter a double value: ");
   scanf("%lf", &d);
   printf("You entered %02i and %0.1f\n", i, d);

   return 0;
}
```

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>`

```
#include <stdio.h>

int main(int argc, char **argv) {
   fprintf(stdout, "My first program in C!\n");
   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]);
      }
   }
}
```

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`

```
clang print_args.c -o print_args
./print_args first second
My first program in C!
Its name is "./print_args"
It has been 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