

Data types: Struct, Union, Enum, Bit Fields

Jan Faigl

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

Lecture 05

PRG(A) – Programming in C

Overview of the Lecture

- Part 1 – Data types
 - Structures – struct
 - Unions
 - Type definition – typedef
 - Enumerations – enum
 - Bit-Fields
- Part 2 – Assignment HW 05
- Part 3 – Coding Examples (optional)
 - Pointer Casting - Print Hex Values
 - Casting Pointer to Array
 - String Sorting
 - Simple Calculator
 - Casting Pointer to Array

K. N. King: chapters 16 and 20

Part I

Data types – Struct, Union, Enum and Bit Fields

Structures, Unions, and Enumerations

- Structure is a collection of values, possibly of different types.
 - It is defined with the keyword **struct**.
 - Structures represent **records** of data **fields**.
- Union is also a collection of values, but its members share the same storage.

Union can store one member at a time, but not all simultaneously.
- Enumeration represents **named integer values**.

Structure Tag

- Declaring a **structure tag** allows to identify a particular structure and avoids repeating all the data fields in the structure variable.

```
struct user_account {  
    int login_count;  
    char username[USERNAME_LEN + 1];  
    int last_login;  
};
```

Notice VLA is not allowed in structure type because the size of the structure needs to be known and determined.

 - After creating the **user_account** tag, variables can be defined as follows.

```
struct user_account user1, user2;
```
- The defined tag is not a type name, therefore it has to be used with the **struct** keyword.
- The new type can be defined using the **typedef** keyword.

```
typedef struct { ... } new_type_name;
```

struct

- Structure **struct** is a finite set of data field members that can be of different type.
- Structure is defined by the programmer as a new data type.
- It allows storing a collection of the related data fields.
- Each structure has a separate **name space** for its members.
- Definition of the compound type (**struct**) variable **user_account**.

```
#define USERNAME_LEN 8  
struct {  
    int login_count;  
    char username[USERNAME_LEN + 1];  
    int last_login; // date as the number of seconds  
    // from 1.1.1970 (unix time)  
} user_account; // variable of the struct defined type
```

Using anonymous structure declaration.
- The declaration follows other variable declaration where **struct {...}** specifies the type and **user_account** the variable name.
- We access the members using the **.** operator, e.g.,

```
user_account.login_count = 0;
```

Example of Defining Structure

- Without definition of the new type (using **typedef**) adding the keyword **struct** before the structure tag is mandatory.

```
struct record {  
    int number;  
    double value;  
};  
record r; /* THIS IS NOT ALLOWED! */  
/* Type record is not known */  
  
typedef struct {  
    int n;  
    double v;  
} item;  
  
struct record r; /* Keyword struct is required */  
item i; /* type item defined using typedef */
```
- Introducing new type by **typedef**, the defined struct type can be used without the **struct** keyword.

```
lec05/struct.c
```

Initialization of the Structure Variables and Assignment Operator

- Structure variables can be initialized in the declaration.
- In C99, we can also use the designated initializers.

```
struct {  
    int login_count;  
    char name[USERNAME_LEN + 1];  
    int last_login;  
} user1 = { 0, "admin", 1477134134 }, //get unix time 'date +%s'  
// designated initializers in C99  
user2 = { .name = "root", .login_count = 128 };  
  
printf("User1 '%s' last login on: %d\n", user1.name, user1.last_login);  
printf("User2 '%s' last login on: %d\n", user2.name, user2.last_login);  
  
user2 = user1; // assignment operator structures  
printf("User2 '%s' last login on: %d\n", user2.name, user2.last_login);  
lec05/structure_init.c
```
- The assignment operator **=** is defined for the structure variables of the same type.

No other operator like != or == is defined for the structures!

Structure Tag and Structure Type

- We define a new structure tag **record** using **struct record**.

```
struct record {  
    int number;  
    double value;  
};
```
- The tag identifier **record** is defined in the name space of the structure tags.

It is not mixed with other type names.
- Using the **typedef**, we introduce a new type named **record**. *Or any other name.*

```
typedef struct record record;
```

 - We define a new identifier **record** as the type name for the **struct record**.
- Structure tag and definition of the type can be combined.

```
typedef struct record {  
    int number;  
    double value;  
} record;  
typedef struct struct_name {  
    int number;  
    double value;  
} type_name;
```

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Example struct – Assignment

- The assignment operator = can be used for two variables of the same struct type.

```

struct record {
    int number;
    double value;
};

typedef struct {
    int n;
    double v;
} item;

struct record rec1 = { 10, 7.12 };
struct record rec2 = { 5, 13.1 };
item i;
print_record(rec1); /* number(10), value(7.120000) */
print_record(rec2); /* number(5), value(13.100000) */
rec1 = rec2;
i = rec1; /* THIS IS NOT ALLOWED! */
print_record(rec1); /* number(5), value(13.100000) */
    
```

lec05/struct.c

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 11 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Example struct – Direct Copy of the Memory

- Having two structure variables of the same size, the content can be directly copied using memory copy. *E.g., using memcpy() from <string.h>.*

```

struct record r = { 7, 21.4};
item i = { 1, 2.3 };
print_record(r); /* number(7), value(21.400000) */
print_item(&i); /* n(1), v(2.300000) */
if (sizeof(i) == sizeof(r)) {
    printf("i and r are of the same size\n");
    memcpy(&i, &r, sizeof(i));
    print_item(&i); /* n(7), v(21.400000) */
}
    
```

- Notice, in the example, the interpretation of the stored data in both structures is identical. In general, it may not be always the case.

lec05/struct.c

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 12 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Size of Structure Variables

- Data representation of the structure may be different from the sum of sizes of the particular data fields (types of the members).

```

struct record {
    int number;
    double value;
};

typedef struct {
    int n;
    double v;
} item;

printf("Size of int: %lu size of double: %lu\n", sizeof(int),
      sizeof(double));
printf("Size of record: %lu\n", sizeof(struct record));
printf("Size of item: %lu\n", sizeof(item));
    
```

Size of int: 4 size of double: 8
 Size of record: 16
 Size of item: 16

lec05/struct.c

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 13 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Size of Structure Variables 1/2

- Compiler might align the data fields to the size of the word (address) of the particularly used architecture. *E.g., 8 bytes for 64-bits CPUs.*
- A compact memory representation can be explicitly prescribed for the clang and gcc compilers by the `__attribute__((packed))`.

```

struct record_packed {
    int n;
    double v;
} __attribute__((packed));
    
```

- Or

```

typedef struct __attribute__((packed)) {
    int n;
    double v;
} item_packed;
    
```

lec05/struct.c

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 14 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Size of Structure Variables 2/2

```

printf("Size of int: %lu size of double: %lu\n",
      sizeof(int), sizeof(double));

printf("record_packed: %lu\n", sizeof(struct record_packed));

printf("item_packed: %lu\n", sizeof(item_packed));
    
```

Size of int: 4 size of double: 8
 Size of record_packed: 12
 Size of item_packed: 12

- The address alignment provides better performance for addressing the particular members at the cost of higher memory requirements.

Eric S. Raymond: The Lost Art of Structure Packing - <http://www.catb.org/esr/structure-packing>.

lec05/struct.c

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 15 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Accessing Members using Pointer to Structure

- The operator `->` can be used to access structure members using a pointer.

```

typedef struct {
    int number;
    double value;
} record_s;

record_s a; // variable a of the type record_s
record_s *p = &a; // variable p of the type pointer (to record_s)

printf("Number %d\n", p->number);
    
```

lec05/struct.c

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 16 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Structure Variables as a Function Parameter

- Structure variable can be pass to a function and also returned.
- We can pass/return the struct itself.

```

struct record print_record(struct record rec) {
    printf("record: number(%d), value(%lf)\n",
          rec.number, rec.value);
    return rec;
}
    
```

- Struct **value** – a new variable is allocated on the stack and data are copied. *Be aware of shallow copy of pointer data fields.*
- Or, as a pointer to a structure.

```

item* print_item(item *v) {
    printf("item: n(%d), v(%lf)\n", v->n, v->v);
    return v;
}
    
```

- Struct **pointer** – only the address is passed to the function. *By passing a pointer, we can save copy of large structures to stack.*

lec05/struct.c

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 17 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Union – variables with Shared Memory

- Union** is a set of members, possibly of different types.
- All the members share the same memory. *Members are overlapping.*
- The size of the union is according to the largest member.
- Union is similar to the `struct` and particular members can be accessed using `.` or `->` for pointers.
- The declaration, union tag, and type definition is also similar to the `struct`.

```

1 union Nums {
2     char c;
3     int i;
4 };
5 Nums nums; /* THIS IS NOT ALLOWED! Type Nums is not known! */
6 union Nums nums;
    
```

lec05/union.c

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 19 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Example union 1/2

- A **union** composed of variables of the types: `char`, `int`, and `double`.

```

1 int main(int argc, char *argv[])
2 {
3     union Numbers {
4         char c;
5         int i;
6         double d;
7     };
8     printf("size of char %lu\n", sizeof(char));
9     printf("size of int %lu\n", sizeof(int));
10    printf("size of double %lu\n", sizeof(double));
11    printf("size of Numbers %lu\n", sizeof(union Numbers));
12
13    union Numbers numbers;
14
15    printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i, numbers.d);
    
```

- Example output:

```

size of char 1
size of int 4
size of double 8
size of Numbers 8
Numbers c: 48 i: 740313136 d: 0.000000
    
```

lec05/union.c

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 20 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Example union 2/2

- The particular members of the `union`

```

1 numbers.c = 'a';
2 printf("\nSet the numbers.c to 'a'\n");
3 printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i, numbers.d);
4
5 numbers.i = 5;
6 printf("\nSet the numbers.i to 5\n");
7 printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i, numbers.d);
8
9 numbers.d = 3.14;
10 printf("\nSet the numbers.d to 3.14\n");
11 printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i, numbers.d);

```

- Example output:

```

Set the numbers.c to 'a'
Numbers c: 97 i: 1374389601 d: 3.140000

Set the numbers.i to 5
Numbers c: 5 i: 5 d: 3.139999

Set the numbers.d to 3.14
Numbers c: 31 i: 1374389535 d: 3.140000

```

lec05/union.c

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 21 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Initialization of Unions

- The union variable can be initialized in the declaration.

```

1 union {
2   char c;
3   int i;
4   double d;
5 } numbers = { 'a' };

```

Only the first member can be initialized

- In C99, we can use the designated initializers.

```

1 union {
2   char c;
3   int i;
4   double d;
5 } numbers = { .d = 10.3 };

```

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 22 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Type Definition – typedef

- The `typedef` can also be used to define new data types, not only structures and unions but also pointers or pointers to functions.
- Example of the data type for pointers to `double` or a new type name for `int`.

```

1 typedef double* double_p;
2 typedef int integer;
3 double_p x, y;
4 integer i, j;

```

- Definition of the new data types (using `typedef`) in header files allows a systematic use of new data types in the whole program.

See, e.g., `<inttypes.h>`

- The main advantage of defining a new type is for complex data types such as structures and pointers to functions.

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 24 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Enumeration Tags and Type Names

- Enum allows to define a subset of integer values and named them.
- We can define enumeration tag similarly to struct and union.

```

enum suit { SPADES, CLUBS, HEARTS, DIAMONDS };
enum s1, s2;

```

- A new enumeration type can be defined using the `typedef` keyword.

```

typedef enum { SPADES, CLUBS, HEARTS, DIAMONDS } suit_t;
suit_t s1, s2;

```

- The enumeration can be considered as an `int` value.

However, we should avoid to directly set enum variable as an integer, as, e.g., value 10 does not correspond to any suit.

- Enumeration can be used in a structure to declare "tag fields".

```

typedef struct {
    enum { SPADES, CLUBS, HEARTS, DIAMONDS } suit;
    enum { RED, BLACK } color;
} card;

```

By using `enum` we clarify meaning of the suit and color data fields.

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 26 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Example – Enumerated Type as Subscript 1/4

- Enumeration constants are integers, and they can be used as subscripts.
- We can also use them to initialize an array of structures.

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 enum weekdays { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };
6
7 typedef struct {
8   char *name;
9   char *abbr; // abbreviation
10 } week_day_s;
11
12 const week_day_s days_en[] = {
13   [MONDAY] = { "Monday", "mon" },
14   [TUESDAY] = { "Tuesday", "tue" },
15   [WEDNESDAY] = { "Wednesday", "wed" },
16   [THURSDAY] = { "Thursday", "thr" },
17   [FRIDAY] = { "Friday", "fri" },
18 };

```

lec05/demo-struct.c

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 27 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Example – Enumerated Type as Subscript 2/4

- We can prepare an array of structures for particular language.
- The program prints the name of the week day and particular abbreviation.

```

19 const week_day_s days_cs[] = {
20   [MONDAY] = { "Ponděli", "po" },
21   [TUESDAY] = { "Úterý", "út" },
22   [WEDNESDAY] = { "Středa", "st" },
23   [THURSDAY] = { "Čtvrtek", "čt" },
24   [FRIDAY] = { "Pátek", "pá" },
25 };
26
27 enum { EXIT_OK = 0, ERROR_INPUT = 101 };
28 int main(int argc, char *argv[], char **envp)
29 {
30   int day_of_week = argc > 1 ? atoi(argv[1]) : 1;
31   if (day_of_week < 1 || day_of_week > 5) {
32     fprintf(stderr, "(EE) File: '%s' Line: %d -- Given day of week out of range\n",
33             _FILE_, _LINE_);
34     return ERROR_INPUT;
35   }
36   day_of_week -= 1; // start from 0

```

lec05/demo-struct.c

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 28 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Example – Enumerated Type as Subscript 3/4

- Detection of the user "locale" is based on the set environment variables.

For simplicity we just detect Czech based on occurrence of 'cs' substring in LC_CTYPE environment variable.

```

35 _Bool cz = 0;
36 while (*envp != NULL) {
37   if (strstr(*envp, "LC_CTYPE") && strstr(*envp, "cs")) {
38     cz = 1;
39     break;
40   }
41   envp++;
42 }
43 const week_day_s *days = cz ? days_cs : days_en;
44
45 printf("%d %s %s\n",
46        day_of_week,
47        days[day_of_week].name,
48        days[day_of_week].abbr);
49 return EXIT_OK;
50 }

```

lec05/demo-struct.c

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 29 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Example – Enumerated Type as Subscript 4/4

```

$ clang demo-struct.c -o demo-struct
$ ./demo-struct
0 Monday mon
$ ./demo-struct 3
2 Wednesday wed
$ LC_CTYPE=cs ./demo-struct 3
2 Středa st
$ LC_CTYPE=cs.UTF-8 ./demo-struct 5; echo $?
4 Pátek pá
0
$ LC_CTYPE=cs.UTF-8 ./demo-struct 9; echo $?
(EE) File: 'demo-struct.c' Line: 32 -- Given day of week out of range
101

```

lec05/demo-struct.c

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 30 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum Bit-Fields

Bitwise Operators

- In low-level programming, such as programs for MCU (micro controller units), we may need to store information as single bits or collection of bits.
- We can use bitwise operators to set or extract particular bit, e.g., a 16-bit unsigned integer variable `uint16_t i`.

- Set the 4 bit of `i`.

```

if ( i & 0x0010 ) ...

```

- Clear the 4 bit of `i`.

```

i &= ~0x0010;

```

- We can give names to particular bits.

```

35 #define RED 1
36 #define GREEN 2
37 #define BLUE 3
38
39 i |= RED; // sets the RED bit
40 i &= ~GREEN; // clears the GREEN bit
41 if (i & BLUE) ... // test BLUE bit

```

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 32 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum **Bit-Fields**

Bit-Fields in Structures

- In addition to bitwise operators, we can declare structures whose members represent bit-fields, e.g., time stored in 16 bits.

```
typedef struct {
    uint16_t seconds: 5; // use 5 bits to store seconds
    uint16_t minutes: 6; // use 6 bits to store minutes
    uint16_t hours: 5; // use 5 bits to store hours
} file_time_t;
```

```
file_time_t time;
```

- We can access the members as a regular structure variable.


```
time.seconds = 10;
```
- The only restriction is that the bit-fields do not have address in the usual sense, and therefore, using address operator & is not allowed.


```
scanf("%d", &time.hours); // NOT ALLOWED!
```

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 33 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum **Bit-Fields**

Bit-Fields Memory Representation

- The way how a compiler handle bit-fields depends on the notion of the **storage units**.
- Storage units are implementation defined (e.g., 8 bits, 16 bits, etc.).
- We can omit the name of the bit-field for padding, i.e., to ensure other bit fields are properly positioned.

```
typedef struct {
    unsigned int seconds: 5;
    unsigned int minutes: 6;
    unsigned int hours: 5;
} file_time_int_s;
// size 4 bytes
printf("Size %lu\n", sizeof(
    file_time_int_s));
```

```
typedef struct {
    unsigned int seconds: 5;
    unsigned int : 0;
    unsigned int minutes: 6;
    unsigned int hours: 5;
} file_time_int_skip_s;
// size 8 bytes because of padding
printf("Size %lu\n", sizeof(
    file_time_int_skip_s));
```

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 34 / 68

Structures – struct Unions Type definition – typedef Enumerations – enum **Bit-Fields**

Bit-Fields Example

```
typedef struct {
    unsigned int seconds: 5;
    unsigned int minutes: 6;
    unsigned int hours: 5;
} file_time_int_s;

void print_time(const file_time_s *t)
{
    printf("%02u:%02u:%02u\n", t->hours, t->minutes, t->seconds);
}

int main(void)
{
    file_time_s time = { // designated initializers
        .hours = 23, .minutes = 7, .seconds = 10 };
    print_time(&time);
    time.minutes += 30;
    print_time(&time);

    // size 2 bytes (for 16 bit short
    printf("Size of file_time_s %lu\n", sizeof(time));
    return 0;
}
```

lec05/bitfields.c 35 / 68

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 33 / 68

Part II

Part 2 – Assignment HW 05

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 34 / 68

HW 05 / HW 5 – Assignment

Topic: Matrix Operations

(B3B36PRG) Mandatory: 2 points; Optional: 2 points; Bonus : 5
 (BAB36PRGA) Mandatory: 3 points; Optional: 4 points; Bonus : 5

- Motivation:** Variable Length Array (VLA) and 2D arrays.
- Goal:** Familiar yourself with VLA and pointers. (optional and bonus) Dynamic allocation and structures.
- Assignment:** <https://cv.fel.cvut.cz/wiki/courses/b3b36prg/hw/hw05>
<https://cv.fel.cvut.cz/wiki/courses/bab36prga/hw/hw05>
 - Read matrix expression – matrices and operators (+, -, and *) from standard input (dimensions of the matrices are provided).
 - Compute the result of the matrix expression or report an error. Dynamic allocation is not needed! Functions for implementing +, *, and - operators are highly recommended!
 - Optional assignment** – compute the matrix expression with respect to the priority of * operator over + and - operators. Dynamic allocation is not need, but it can be helpful.
 - Bonus assignment** – Read declaration of matrices prior the matrix expression. Dynamic allocation can be helpful, structures are not needed but can be helpful.
- (B3B36PRG) **Deadline: 15.04.2023, 23:59 AoE** (bonus 26.5.2023, 23:59 CEST)
- (BAB36PRGA) **Deadline: 22.04.2023, 23:59 AoE** (bonus 26.5.2023, 23:59 CEST)

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 37 / 68

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 35 / 68

Part III

Part 3 – Coding Examples (optional)

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 36 / 68

Coding Example – Print Hex Values

- Representation of the **float** values.
 - Value 85.125 is 0x42aa4000.
 - Value 0.1 is 0x3dcccc but encoded 0x3dcccccd.
- Implement a function to print a hex representation of a float value.
- Access to a float value as a sequence of bytes and print individual bytes as hex values using "%02x" in `printf()`.
 - Use addressing operator & to get variable address.
 - Type case to get a pointer to char (a single byte).
 - Use indirect addressing operator * to access to the variable at the address stored in the pointer variable.
- Access to a float value as a sequence of bytes and print individual bytes as hex values using "%02x" in `printf()`.

```
#include <stdio.h>

void print_float_hex(float v);

int main(void)
{
    print_float_hex(85.125);
    print_float_hex(0.1);
    return 0;
}

void print_float_hex(float v)
{
    ...
}
```

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 40 / 68

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 37 / 68

Coding Example – Print Hex Values – Implementation 1/3

- Retrieve address of variable **float v** by &v.
- We need access values at the address &v as bytes; therefore, we type cast it to a pointer to char value(s).


```
unsigned char *p = (unsigned char*)&v;
```
- The value at the address stored in **p** can be accessed by the indirect addressing operator *p.
- We can advance the next address by incrementing the value stored in **p**, e.g., `p = p + 1;`. Because it is a pointer to char, the increment is about sizeof(char), i.e., by 1. It is the pointer arithmetic.
- However, the printed values are in the reversed order than the expected order 0x42aa4000 and 0x3dcccccd.

```
int main(void)
{
    print_float_hex(85.125);
    print_float_hex(0.1);
    ...
    void print_float_hex(float v)
    {
        unsigned char *p = (unsigned char*)&v;
        printf("Value %13.10f is 0x", v);
        for (int i = 0; i < 4; ++i, p = p + 1) {
            printf("%02x", *p); // or use p[i]
        }
        putchar('\n');
    }

    $ clang floats.c -o floats && ./floats
    Value 85.1250000000 is 0x0040aa42
    Value 0.1000000015 is 0xcdcccc3d
```

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 41 / 68

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 38 / 68

Coding Example – Print Hex Values – Implementation 2/3

- Expected hexadecimal representation of the values 85.125 and 0.1 is 0x42aa4000 and 0x3dcccccd but the printed values are 0x0040aa42 and 0xcdcccc3d, respectively.
- It is because of the way how multi-byte values are stored in the memory. For the used architecture (amd64), it is little endian.
- Thus, we need to detect the endianness. <https://en.wikipedia.org/wiki/Endianness>
- E.g., using a function `_Bool is_big_endian(void);`
- and print values in the reversed order.

```
void print_float_hex(float v)
{
    const _Bool big_endian = is_big_endian();
    // cast pointer to float to pointer to char
    unsigned char *p = (unsigned char*)&v
        + (big_endian ? 0 : 3);
    printf("Value %13.10f is 0x", v);
    for (int i = 0; i < 4; ++i) {
        printf("%02x",
            *(big_endian ? p++ : p--));
    }
    printf("\n");
}

$ clang floats.c -o floats && ./floats
Value 85.1250000000 is 0x42aa4000
Value 0.1000000015 is 0x3dcccccd
```

Jan Faigl, 2023 PRG(A) – Lecture 05: Data types 42 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Print Hex Values – Implementation 3/3

- The detection of the endianness can be based on various techniques.
- Intuitively, we need to store a defined value with all zeros but one byte non-zero.
- We can take advantage of the `union` type that allows different views on the identical memory block.
 - Define an integer variable with the specified size of four bytes, e.g., `uint32_t` from `stdint.h` library.
 - Set the value of `0x01 00 00 00` to the variable.
 - Check the first byte of the memory representation, if it is zero or one.

```
#include <stdint.h>

_Bool is_big_endian(void)
{
    union {
        uint32_t i;
        char c[4];
    } e = { 0x01000000 };
    return e.c[0];
}
```

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 43 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Array and Pointer to Function 1/4

- Implement a program that creates an array of random integer values using `rand()` function from `stdlib.h`.
- The integer values are limited to `MAX_NUM` set to, e.g., 20, by `#define MAX_NUM 20`.
- The default number can be adjusted at the compile time – `clang -DLEN=10 program.c`.
- The array is printed to `stdout`. *Print function.*
- The array is sorted using `qsort()` from `stdlib.h`. *Become familiar with `man qsort`.*
- The sorted array is printed to `stdout`.
- The program is then enhanced by processing program arguments to define the no. of values as the first program argument using `atoi()`.

```
#ifndef LEN
#define LEN 5
#endif

#define MAX_NUM 20

void fill_random(size_t l, int a[l]);
void print(const char *s, size_t l, int a[l]);

int main(void)
{
    int a[LEN]; // allocate the array
    fill_random(LEN, a); // fill the array
    print("Array random: ", LEN, a);
    // TODO call qsort
    print("Array sorted: ", LEN, a);
    return 0;
}
```

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 45 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Array and Pointer to Function 2/4

```
void fill_random(size_t l, int a[l])
{
    for (size_t i = 0; i < l; ++i) {
        a[i] = rand() % MAX_NUM;
    }
}

void print(const char *s, size_t l, int a[l])
{
    if (s) {
        printf("%s", s);
    }
    for (size_t i = 0; i < l; ++i) {
        printf("%s%d", i > 0 ? " " : "", a[i]);
    }
    putchar('\n');
}

void qsort(
    void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *)
);

// base is the pointer to the initial member.
// nmemb is the no. of members.
// size is the size of each member.
// compar is a pointer to the comparison function.

int compare(const void *ai, const void *bi)
{
    const int *a = (const int*)ai;
    const int *b = (const int*)bi;
    //ascending
    return *a == *b ? 0 : (*a < *b ? -1 : 1);
}

// Change the order to descending.
```

- See `man qsort` for `qsort` synopsis.

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 46 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Array and Pointer to Function 3/4

- Use the function name as the pointer to the function.


```
int compare(const void *, const void *);

int main(void)
{
    int a[LEN]; // do not initialize
    fill_random(LEN, a);
    print("Array random: ", LEN, a);
    qsort(a, LEN, sizeof(int), compare);
    print("Array sorted: ", LEN, a);
    return 0;
}
```
- Compile and run if the compilation is successful using `shell logical and` operator `&&`.


```
$ clang sort.c -o sort && ./sort
Array random: 13 17 18 15 12
Array sorted: 12 13 15 17 18
```
- Use compiler flag `-DLEN=10` to define the array length 10.


```
$ clang -DLEN=10 sort.c -o sort && ./sort
Array random: 13 17 18 15 12 3 7 8 18 10
Array sorted: 3 7 8 10 12 13 15 17 18 18
```

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 47 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Array and Pointer to Function 4/4

- Extend `main()` to pass program arguments.
- Define an error value.


```
enum { ERROR = 100 };

int main(int argc, char *argv[])
{
    const size_t len = argc > 1 ?
        atoi(argv[1]) : LEN;
    if (len > 0) {
        int a[len];
        fill_random(len, a);
        print("Array random: ", len, a);
        qsort(a, len, sizeof(int), compare);
        print("Array sorted: ", len, a);
    }
    return len > 0 ? EXIT_SUCCESS : ERROR;
}
```
- We use the **Variable Length Array (VLA)**, which length is determined during the runtime.


```
$ clang sort-vla.c -o sort && ./sort
Array random: 13 17 18 15 12 3
Array sorted: 3 12 13 15 17 18

$ clang sort-vla.c -DLEN=7 -o sort && ./sort
Array random: 13 17 18 15 12 3 7
Array sorted: 3 7 12 13 15 17 18

$ clang sort-vla.c -o sort && ./sort 11
Array random: 13 17 18 15 12 3 7 8 18 10 9
Array sorted: 3 7 8 10 12 13 15 17 18 18 19
```
- Be aware the size of the array `a` is limited by the size of the `stack`, see `ulimit -s`.

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 48 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – String Sorting 1/5

- Implement a program that sorts program arguments lexicographically using `strcmp` (from `string.h`) and `qsort` (from `stdlib.h`).
- Print the arguments. *Print function.*
- Copy the passed `argv` to newly allocated memory on the heap to avoid changes in `argv`.
 - Exit with -1 if allocation fails.
 - Copy strings using `strncpy`. *My malloc function. Copy and copy strings functions.*
- Sort the copied array of strings with the help of `strcmp`. *String compare function.*
- Release the allocated memory. *Release function.*

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

void print(int n, char *strings[n]);

char* copy(const char *str);
char** copy_strings(int n, char *strings[n]);

void* my_malloc(size_t size);
void release(int n, char **strings);

int string_compare(
    const void *p1, const void *p2);

enum { EXIT_OK = 0, EXIT_MEM = -1 };

int main(int argc, char *argv[]);
```

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 50 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – String Sorting 2/5

- Print function directly iterates over strings.


```
void print(int n, char *strings[n])
{
    for (int i = 0; i < n; ++i) {
        printf("%3d. \"%s\"\n", i, strings[i]);
    }
}
```
- Allocate array of pointers to char.


```
char** copy_strings(int n, char *strings[n])
{
    char** ret = my_malloc(n * sizeof(char*));
    for (int i = 0; i < n; ++i) {
        ret[i] = copy(strings[i]);
    }
    return ret;
}
```
- Copy call `my_malloc` and use `strncpy`.


```
char* copy(const char *str)
{
    char *ret = NULL;
    if (str) {
        size_t len = strlen(str);
        ret = my_malloc(len + 1); // +1 for '\0'
        strncpy(ret, str, len + 1); // +1 for '\0'
    }
    return ret;
}
```

 - The length of the string (by `strlen`) is without the null terminating `'\0'`.
 - The copy of the string content needs to include the null terminating character as well.

We take advantage that the allocation succeeds, or the program terminates with an error.

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 51 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – String Sorting 3/5

- Dynamic allocation calls `malloc` and terminates the program on error.


```
void* my_malloc(size_t size)
{
    void *ret = malloc(size);
    if (!ret) {
        fprintf(stderr,
            "ERROR: Mem allocation error!\n");
        exit(EXIT_MEM);
    }
    return ret;
}
```
- The dynamically allocated array of pointers to (dynamically allocated) strings needs releasing the strings and then the array itself.


```
void release(int n, char **strings)
{
    if (strings && *strings)
        return;

    for (int i = 0; i < n; ++i) {
        if (strings[i]) {
            free(strings[i]); // free string
        }
        free(strings); // free array of pointers
    }
}
```

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 52 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – String Sorting 4/5

- Synopsis of the `qsort` function, see `man qsort`.


```
void qsort(void *base, size_t nmemb, size_t size,
    int (*compar)(const void *, const void *)
);
```

It passes pointers to the array elements as pointers to constant values.
- We call `qsort` on an array of pointers to strings, which are pointers to char.


```
char **strings = copy_strings(n, argv);
qsort(strings, n, sizeof(char*), string_compare);
```
- We cast the pointer to void as a pointer to pointer to char for accessing the string.


```
int string_compare(const void *p1, const void *p2)
{
    char * const *s1 = p1; // qsort passes a pointer to the array item (string)
    char * const *s2 = p2;
    return strcmp(*s1, *s2);
}
```

Jan Faijl, 2023 PRG(A) – Lecture 05: Data types 53 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – String Sorting 5/5

- Call `qsort` on array of pointers.

```
int main(int argc, char *argv[])
{
    int ret = EXIT_OK;
    const int n = argc;
    printf("Arguments:\n");
    print(argc, argv);

    char **strings = copy_strings(n, argv);
    qsort(
        strings, n,
        sizeof(char*), string_compare
    );

    printf("\n Sorted arguments:\n");
    print(n, strings);
    release(n, strings);
    return ret;
}
```

PRG(A) – Lecture 05: Data types 54 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Simple Calculator 1/6

- Implement a calculator that processes an input string containing expression with interger values and operators '+', '-', '*'.
Sum, sub, and mult functions.
- It reports error and return error values 100 if value is not an interger and 101 in the case of unsupported operator.
- Use pointer to operation functions.
- Process the input step-by-step, avoid reading the whole input, print partial results.
- Handle all possible errors.
 - There must be by at least single interger value.
 - If an operator is given, it must be valid and there must be the second operand.
 - If end-of-file (input), and the operator is not given, print the result.

```
enum status { EXIT_OK = 0, ERROR_INPUT = 100,
             ERROR_OPERATOR = 101 };
enum status print(enum status error);
int main(int argc, char *argv[])
{
    enum status ret = EXIT_OK;
    ...
    return print(ret);
}
enum status print(enum status error)
{
    if (error == ERROR_INPUT) {
        fprintf(stderr, "ERROR: Input value\n");
    } else if (error == ERROR_OPERATOR) {
        fprintf(stderr, "ERROR: Operator\n");
    }
    return error;
}
```

PRG(A) – Lecture 05: Data types 56 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Simple Calculator 2/6

- Implement a calculator that processes an input string containing expression with interger values and operators '+', '-', '*'.
Sum, sub, and mult functions.
- It reports error and return error values 100 if value is not an interger and 101 in the case of unsupported operator.
- Use pointer to operation functions.
- Process the input step-by-step, avoid reading the whole input, print partial results.
- Handle all possible errors.
 - There must be by at least single interger value.
 - If an operator is given, it must be valid and there must be the second operand.
 - If end-of-file (input), and the operator is not given, print the result.

```
int sum(int a, int b); // return a + b
int sub(int a, int b); // return a - b
int mult(int a, int b); // return a * b
//define a pointer to a function
typedef int (*ptr)(int, int);
//typedef ptr is needed for the return value
ptr getop(const char *op)
{
    int (*operation)(int, int) = NULL;
    if (op[0] == '+') {
        operation = sum;
    } else if (op[0] == '-') {
        operation = sub;
    } else if (op[0] == '*') {
        operation = mult;
    }
    return operation;
}
```

PRG(A) – Lecture 05: Data types 57 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Simple Calculator 3/6

- Implement a calculator that processes an input string containing expression with interger values and operators '+', '-', '*'.
Sum, sub, and mult functions.
- It reports error and return error values 100 if value is not an interger and 101 in the case of unsupported operator.
- Use pointer to operation functions.
- Process the input step-by-step, avoid reading the whole input, print partial results.
- Handle all possible errors.
 - There must be by at least single interger value.
 - If an operator is given, it must be valid and there must be the second operand.
 - If end-of-file (input), and the operator is not given, print the result.

```
int r = 1; //the first v1
char opstr[2] = {}; //store the operator
ptr op = NULL; // function pointer
int v2; //store the second operand
while (r == 1 && ret == EXIT_OK) {
    r = (op = readop(opstr, &ret)) ? 1 : 0;
    // operator is valid and second operand read
    int v3 = op(v1, v2);
    printf("%3d %s %3d = %3d\n",
        v1, opstr, v2, v3);
    v1 = v3; //shift the results
} else if (!op) { // no operator
    printf("Result: %3d\n", v1);
    r = 0;
} else if (r != 1) { //no operand
    ret = ERROR_INPUT;
} //end of while
}
```

PRG(A) – Lecture 05: Data types 58 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Simple Calculator 4/6

- Implement a calculator that processes an input string containing expression with interger values and operators '+', '-', '*'.
Sum, sub, and mult functions.
- It reports error and return error values 100 if value is not an interger and 101 in the case of unsupported operator.
- Use pointer to operation functions.
- Process the input step-by-step, avoid reading the whole input, print partial results.
- Handle all possible errors.
 - There must be by at least single interger value.
 - If an operator is given, it must be valid and there must be the second operand.

```
enum status ret = EXIT_OK;
int v1;
int r = scanf("%d", &v1) == 1;
ret = r == 0 ? ERROR_INPUT : ret;
if (ret == EXIT_OK) {
    ret = process(ret, v1);
}
...
ptr readop(char *opstr, enum status *error)
{
    ptr op = NULL; // pointer to a function
    int r = scanf("%s", opstr);
    if (r == 1) {
        *error = (op = getop(opstr)) ? *error :
            ERROR_OPERATOR;
    } // else end-of-file
    return op;
}
```

PRG(A) – Lecture 05: Data types 59 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Simple Calculator 5/6

```
enum status process(enum status ret, int v1)
{
    int r = 1; //the first operand is given in v1
    char opstr[2] = {}; //store the operator
    ptr op = NULL; // function pointer to operator
    int v2; //store the second operand
    while (r == 1 && ret == EXIT_OK) {
        r = (op = readop(opstr, &ret)) ? 1 : 0; // operand read successfully
        if (r == 1 && (r = scanf("%d", &v2)) == 1) { // while ends for r == 0 or r == -1
            int v3 = op(v1, v2);
            printf("%3d %s %3d = %3d\n", v1, opstr, v2, v3);
            v1 = v3; //shift the results
        } else if (!op) { // no operator in the input
            printf("Result: %3d\n", v1); //print the final results
            r = 0;
        } else if (r != 1) { //no operand on the input
            ret = ERROR_INPUT;
        }
    } //end of while
    return ret;
}
```

PRG(A) – Lecture 05: Data types 60 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Simple Calculator 6/6

- Example of program execution.

```
$ clang calc.c -o calc
$ echo "1 + 2 * 6 - 2 * 3 + 19" | ./calc
1 + 2 = 3
3 * 6 = 18
18 - 2 = 16
16 * 3 = 48
48 + 19 = 67
Result: 67
$ echo "1 + 2 *" | ./calc; echo $?
1 + 2 = 3
ERROR: Input value
100
$echo "1 + 2 a" | ./calc; echo $?
1 + 2 = 3
Result: 3
ERROR: Operator
```

PRG(A) – Lecture 05: Data types 61 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Casting Pointer to Array 1/4

- Allocate array of the size `ROWS × COLS` and fill it with random interger values with up to two digits, and print the values are an array.
- Implement fill and print functions.
- Implement print function to print matrix of the size `rows × cols`.
- Cast the array of `int` values into `m` - a pointer of arrays of the size `cols`.
- Pass `m` to the function that prints the 2D array (matrix) with `cols` columns.

```
#define MAX_VALUE 100
#define ROWS 3
#define COLS 4
void fill(int n, int *v);
void print_values(int n, int *a);
int main(int argc, char *argv[])
{
    const int n = ROWS * COLS;
    int array[n];
    int *p = array;
    fill(n, p);
    print_values(n, p);
    return 0;
}
```

PRG(A) – Lecture 05: Data types 63 / 68

Pointer Casting - Print Hex Values Casting Pointer to Array String Sorting Simple Calculator Casting Pointer to Array

Coding Example – Casting Pointer to Array 2/4

- Allocate array of the size `ROWS × COLS` and fill it with random interger values with up to two digits, and print the values are an array.
- Implement fill and print functions.
- Implement print function to print matrix of the size `rows × cols`.
- Cast the array of `int` values into `m` - a pointer of arrays of the size `cols`.
- Pass `m` to the function that prints the 2D array (matrix) with `cols` columns.

```
void fill(int n, int *v)
{
    for (int i = 0; i < n; ++i) {
        v[i] = rand() % MAX_VALUE;
    }
}
void print_values(int n, int *a)
{
    for (int i = 0; i < n; ++i) {
        printf("%s%i",
            (i > 0 ? " " : ""),
            a[i]
        );
    }
    putchar('\n');
}
```

PRG(A) – Lecture 05: Data types 64 / 68

Coding Example – Casting Pointer to Array 3/4

- Allocate array of the size `ROWS × COLS` and fill it with random integer values with up to two digits, and print the values are an array.
- Implement fill and print functions.
- Implement print function to print matrix of the size `rows × cols`.
- Cast the array of `int` values into `m` - a pointer of arrays of the size `cols`.
- Pass `m` to the function that prints the 2D array (matrix) with `cols` columns.

```
void print(int rows, int cols, int m[][cols])
{
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < cols; ++c) {
            printf("%3i", m[r][c]);
        }
        putchar('\n');
    }
}
```

- The number of columns is mandatory to determine the address of the cell `m[r][c]` in the 2D array (matrix) `m`.
- The pointer `m` can refer to arbitrary number of rows.

Coding Example – Casting Pointer to Array 4/4

- Allocate array of the size `ROWS × COLS` and fill it with random integer values with up to two digits, and print the values are an array.
- Implement fill and print functions.
- Implement print function to print matrix of the size `rows × cols`.
- Cast the array of `int` values into `m` - a pointer of arrays of the size `cols`.
- Pass `m` to the function that prints the 2D array (matrix) with `cols` columns.
Try to print the array as matrix with `cols` columns and `rows` columns that is as matrix with `rows×cols` and `cols×rows`, respectively.

```
#define MAX_VALUE 100
#define ROWS 3
#define COLS 4
...
void print(int rows, int cols, int m[][cols]);
int main(int argc, char *argv[])
{
    const int n = ROWS * COLS;
    int array[n];
    int *p = array;
    int (*m)[COLS] = (int(*)[COLS])p;
    printf("\nPrint as matrix %d x %d\n",
        ROWS, COLS);
    print(ROWS, COLS, m);
    return 0;
}
```

Summary of the Lecture

Topics Discussed

- Data types
 - Structure variables
 - Unions
 - Enumeration
 - Type definition
 - Bit-Fields
- Next: Input/output operations and standard library