

# Data types: Struct, Union, Enum, Bit Fields. Preprocessor and Building Programs

Jan Faigl

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

Lecture 05

B3B36PRG – C Programming Language

## Overview of the Lecture

- Part 1 – Data types
  - Structures – struct
  - Unions
  - Type definition – typedef
  - Enumerations – enum
  - Bit-Fields
- Part 2 – Preprocessor and Building Programs
  - Organization of Source Files
  - Preprocessor
  - Building Programs
- Part 3 – Assignment HW 05

*K. N. King: chapters 16 and 20*

*K. N. King: chapters 10, 14, and 15*

## 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;  
};
```

  - After creating the **user\_account** tag, variables can be declared

```
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 as

```
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 // Using anonymous structure declaration  
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
```
- 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 */  
  
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

```
typedef struct {  
    int number;  
    double value;  
} item;  
  
item i;
```

## 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

- Using **struct record** we defined a new structure tag **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**

```
typedef struct record record;
```

  - We define a new global 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;
```

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 Faigl, 2021 B3B36PRG – Lecture 05: Data types 11 / 53

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 the <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 this 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 Faigl, 2021 B3B36PRG – Lecture 05: Data types 12 / 53

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 Faigl, 2021 B3B36PRG – Lecture 05: Data types 13 / 53

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

### Size of Structure Variables 1/2

- Compiler may 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 Faigl, 2021 B3B36PRG – Lecture 05: Data types 14 / 53

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 Faigl, 2021 B3B36PRG – Lecture 05: Data types 15 / 53

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;
record_s *p = &a;

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

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 16 / 53

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

```

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

or as a pointer to a structure

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

Passing the structure by
    value, a new variable is allocated on the stack and data are copied
    pointer only the address is passed to the function
    
```

*Be aware of shallow copy of pointer data fields.*

lec05/struct.c

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 17 / 53

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;
    
```

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 19 / 53

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 Faigl, 2021 B3B36PRG – Lecture 05: Data types 20 / 53

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: %f\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: %f\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: %f\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, 2021    B3B36PRG – Lecture 05: Data types    21 / 53

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, 2021    B3B36PRG – Lecture 05: Data types    22 / 53

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 structures
- 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;

```
- The usage is identical to the default data types

```

1 double *x, *y;
2 int 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, 2021    B3B36PRG – Lecture 05: Data types    24 / 53

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, 2021    B3B36PRG – Lecture 05: Data types    26 / 53

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

### Example – Enumerated Type as Subscript 1/3

- 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, 2021    B3B36PRG – Lecture 05: Data types    27 / 53

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

### Example – Enumerated Type as Subscript 2/3

- 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] = { "Pondeli", "po" },
21     [TUESDAY] = { "Utery", "ut" },
22     [WEDNESDAY] = { "Streda", "st" },
23     [THURSDAY] = { "Ctvrtek", "ct" },
24     [FRIDAY] = { "Patek", "pa" },
25 };
26
27 int main(int argc, char *argv[], char **envp)
28 {
29     int day_of_week = argc > 1 ? atoi(argv[1]) : 1;
30     if (day_of_week < 1 || day_of_week > 5) {
31         fprintf(stderr, "(EE) File: '%s' Line: %d -- Given day of week out of range\n",
32             __FILE__, __LINE__);
33         return 101;
34     }
35     day_of_week --; // start from 0

```

lec05/demo-struct.c

Jan Faigl, 2021    B3B36PRG – Lecture 05: Data types    28 / 53

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

### Example – Enumerated Type as Subscript 3/3

- 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 0;

```

lec05/demo-struct.c

Jan Faigl, 2021    B3B36PRG – Lecture 05: Data types    29 / 53

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
- To set or extract particular bit, we can use bitwise operators, e.g., a 16-bit unsigned integer variable `uint16_t i`
  - Set the 4 bit of `i`

```

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, 2021    B3B36PRG – Lecture 05: Data types    31 / 53

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;

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

```

file_time_t time;
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, 2021    B3B36PRG – Lecture 05: Data types    32 / 53

### 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));
```

### 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;
}
```

## Part II Preprocessor and Building Programs

### Variables – Scope and Visibility

- **Local variables**
  - A variable declared in the body of a function is the **local variable**
  - Using the keyword **static** we can declared **static local variables**
  - Local variables are visible (and accessible) only within the function
- **External variables** (global variables)
  - Variables declared outside the body of any function
  - They have **static storage duration**; the value is stored as the program is running
  - External variable has **file scope**, i.e., it is visible from its point of the declaration to the end of the enclosing file
    - We can refer to the external variable from other files by using the **extern** keyword
    - In a one file, we define the variable, e.g., as **int var**;
    - In other files, we declare the external variable as **extern int var**;
  - We can restrict the visibility of the **global variable** to be within the single file only by the **static** keyword

### Organizing C Program

- Particular source files can be organized in many ways
- A possible ordering of particular parts can be as follows:
  1. #include directives
  2. #define directives
  3. Type definitions
  4. Declarations of external variables
  5. Prototypes for functions other than **main()** (if any)
  6. Definition of the **main()** function (if any)
  7. Definition of other functions

### Header Files

- Header files provide the way how to share defined macros, variables, and use functions defined in other modules (source files) and libraries
- **#include** directive has two forms
  - **#include <filename>** – to include header files that are searched from system directives
  - **#include "filename"** – to include header files that are searched from the current directory
- The places to be searched for the header files can be altered, e.g., using the command line options such as **-Ipath**
- It is not recommended to use brackets for including own header files
- It is also not recommended to use absolute paths
 

*Neither windows nor unix like absolute paths*

### Example of Sharing Macros and Type Definition, Function Prototypes and External Variables

- Let have three files **graph.h**, **graph.c**, and **main.c**
- We like to share macros and types, and also functions and external variables defined in **graph.c** in **main.c**

```
graph.h
#define GRAPH_SIZE 1000
typedef struct {
    ...
} edget_s;
typedef struct {
    edget_s *edges;
    int size;
} graph_s;
// make the graph_global extern
extern graph_s graph_global;
// declare function prototype
graph_s* load_graph(const char *filename);

graph.c
#include "graph.h"
graph_s graph_global = { NULL, GRAPH_SIZE };
...
graph_s* load_graph(const char *filename)
{
    ...
}

main.c
#include "graph.h"
int main(int argc, char *argv[])
{
    // we can use function from graph.c
    graph_s *graph = load_graph(...);
    // we can also use the global variable
    // declared as extern in the graph.h
    if (global_graph.size != GRAPH_SIZE) { ...
```

### Protecting Header Files

- Header files can be included from other header files
- It may happen that the same type can be defined multiple times due to including header files
- We can protect header files from multiple includes by using the preprocessor macros
 

```
#ifndef GRAPH_H
#define GRAPH_H
...
header file body here
// it is processed only if GRAPH_H is not defined
// therefore, after the first include,
// the macro GRAPH_H is defined
// and the body is not processed during therepeated includes
#endif
```

### Macros

- Macro definitions – **#define**
  - The macros can be parametrized, i.e., function-like macros
  - Already defined macros can be undefined by the **#undef** command
- File inclusion – **#include**
- Conditional compilation – **#if**, **#ifdef**, **#ifndef**, **#elif**, **#else**, **#endif**
- Miscellaneous directives
  - **#error** – produces error message, e.g., combined with **#if** to test sufficient size of **MAX\_INT**
  - **#line** – alter the way how lines are numbered (**\_\_LINE\_\_** and **\_\_FILE\_\_** macros)
  - **#pragma** – provides a way to request a special behaviour from the compiler
 

*C99 introduces **\_Pragma** operator used for "destringing" the string literals and pass them to **#pragma** operator.*

Organization of Source Files Preprocessor Building Programs

## Predefined Macros

- There are several predefined macros that provide information about the compilation and compiler as integer constant or string literal
  - `__LINE__` – Line number of the file being compiled (processed)
  - `__FILE__` – Name of the file being compiled
  - `__DATE__` – Date of the compilation (in the form "Mmm dd yyyy")
  - `__TIME__` – Time of the compilation (in the form "hh:mm:ss")
  - `__STDC__` – 1 if the compiler conforms to the C standard (C89 or C99)
- C99 introduces further macros, e.g.,
  - `__STDC_VERSION__` – Version of C standard supported
    - For C89 it is 199409L
    - For C99 it is 199901L
- It also introduces identifier `__func__` which provides the name of the actual function
 

*It is actually not a macro, but behaves similarly*

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 44 / 53

Organization of Source Files Preprocessor Building Programs

## Defining Macros Outside a Program

- We can control the compilation using the preprocessor macros
- The macros can be defined outside a program, e.g., during the compilation by passing particular arguments to the compiler
- For `gcc` and `clang` it is the `-D` argument, e.g.,
  - `gcc -DDEBUG=1 main.c` – define macro `DEBUG` and set it to 1
  - `gcc -DNDEBUG main.c` – define `NDEBUG` to disable `assert()` macro

*See man assert*
- The macros can be also undefined, e.g., by the `-U` argument
- Having the option to define the macros by compiler options, we can control the compilation process according to the particular environment and desired target platform

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 45 / 53

Organization of Source Files Preprocessor Building Programs

## Compiling and Linking

- Programs composed of several modules (source files) can be build by an individual compilation of particular files, e.g., using `-c` option of the compiler
- Then, all object files can be linked to a single binary executable file
- Using the `-l` option, we can add a particular `lib` library
- E.g., let have source files `module1.c`, `module2.c`, and `main.c` that also depends on the `math` library (`-lm`)
  - The program can be build as follows
 

```
clang -c module1.c -o module1.o
clang -c module2.c -o module2.o
clang -c main.c -o main.o
```

`clang main.o module2.o module1.o -lm -o main`

*Be aware that the order of the files is important for resolving dependencies! It is incremental, i.e., only the function needed in first modules are linked from the other modules.*

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 47 / 53

Organization of Source Files Preprocessor Building Programs

## Makefile

- Some building system may be suitable for project with several files
- One of the most common tools is the GNU `make` or the `make`

*Notice, there are many building systems that may provide different features, e.g., designed for the fast evaluation of the dependencies like `ninja`*
- For `make`, the building rules are written in the `Makefile` files
 

<http://www.gnu.org/software/make/make.html>
- The rules define targets, dependencies, and action to build the targets based on the dependencies
 

```
target : dependencies          colon
          action                tabulator
```
- Target can be symbolic name or file name
 

```
main.o : main.c
      clang -c main.c -o main.o
```
- The receipt to build the program can be simple, e.g., using explicitly the file names and compiler options
 

The main advantage of the Makefiles is flexibility arising from unified variables, internal make variables, and templates as most of the sources can be compiled in pretty much similar way.

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 48 / 53

Organization of Source Files Preprocessor Building Programs

## Example Makefile

- Pattern rule for compiling source files `.c` to object files `.o`
- Wildcards are used to compile all source files in the directory
 

*Can be suitable for small project. In general, explicit listings of the files is more appropriate.*

```
CC=ccache $(CC)
CFLAGS+=-O2
OBSJ=$(patsubst %.c,%.o,$(wildcard *.c))
TARGET=program
bin: $(TARGET)
$(OBSJ): %.o: %.c
$(CC) -c $(CFLAGS) $(CPPFLAGS) -o $@
$(TARGET): $(OBSJ)
$(CC) $(OBSJ) $(LDFLAGS) -o $@
clean:
$(RM) $(OBSJ) $(TARGET)
CC=clang make vs CC=gcc make
```
- The order of the files is important during the linking!

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 49 / 53

Organization of Source Files Preprocessor Building Programs

## Part III

### Part 3 – Assignment HW 05

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 50 / 53

Organization of Source Files Preprocessor Building Programs

## HW 05 – Assignment

Topic: **Matrix Operations** Mandatory: 2 points; Optional: 2 points; Bonus : 5

- Motivation: Variable Length Array (VLA) and 2D arrays
- Goal: Familiar yourself with VLA and pointers
- Assignment:
  - Eventually with dynamic allocation and structures
 

<https://cw.fel.cvut.cz/wiki/courses/b3b36prg/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.*
- Deadline: 27.03.04.2021, 23:59:59 PDT, Bonus part 22.05.2021
 

*PDT – Pacific Daylight Time*

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 51 / 53

Organization of Source Files Preprocessor Building Programs

## Summary of the Lecture

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 52 / 53

Organization of Source Files Preprocessor Building Programs

## Topics Discussed

- Data types
  - Structure variables
  - Unions
  - Enumeration
  - Type definition
  - Bit-Fields
- Building Programs
  - Variables and their scope and visibility
  - Organizing source codes and using header files
  - Preprocessor macros
  - Makefiles
- Next: Input/output operations and standard library

Jan Faigl, 2021 B3B36PRG – Lecture 05: Data types 53 / 53