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 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**

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
- Declaration of the `struct` variable is

```
#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
```

- 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;`

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[USENAME_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

- 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.

  - 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;`

Example of Defining Structure

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

```
struct record {             typedef struct {
   int number;                 int n;
   double value;               double v;
};                          } item;

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

lec05/struct.c

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;
```

Example struct – Assignment

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

```c
struct record {                     typedef struct {
   int number;                         int n;
   double value;                       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

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>

```c
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

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)

```c
struct record {                     typedef struct {
   int number;                         int n;
   double value;                       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

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))`

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

- Or

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

lec05/struct.c

Size of Structure Variables 2/2

```c
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
```

lec05/struct.c

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

http://www.catb.org/esr/structure-packing

Accessing Members using Pointer to Structure

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

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

record_s a;
record_s *p = &a;

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


Structure Variables as a Function Parameter

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

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

- or as a pointer to a structure

```c
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

    Be aware of shallow copy of pointer data fields.
  - **pointer** only the address is passed to the function

lec05/struct.c

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

```c
union Nums {
   char c;
   int i;
};
Nums nums; /* THIS IS NOT ALLOWED! Type Nums is not known! */
union Nums nums;
```


Example union 1/2

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

```c
int main(int argc, char *argv[])
{
   union Numbers {
      char c;
      int i;
      double d;
   };
   printf("size of char %lu\n", sizeof(char));
   printf("size of int %lu\n", sizeof(int ));
   printf("size of double %lu\n", sizeof(double));
   printf("size of Numbers %lu\n", sizeof(union Numbers));

   union Numbers numbers;

   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

Example union 2/2

- The particular members of the union

```
numbers.c = 'a';
printf("\nSet the numbers.c to 'a'\n");
printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i, numbers.d);

numbers.i = 5;
printf("\nSet the numbers.i to 5\n");
printf("Numbers c: %d i: %d d: %lf\n", numbers.c, numbers.i, numbers.d);

numbers.d = 3.14;
printf("\nSet the numbers.d to 3.14\n");
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

Initialization of Unions

- The union variable can be initialized in the declaration

```
union {
   char c;
   int i;
   double d;
} numbers = { 'a' };
```

Only the first member can be initialized

- In C99, we can use the designated initializers

```
union {
   char c;
   int i;
   double d;
} numbers = { .d = 10.3 };
```


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:

```
typedef double* double_p;
typedef int integer;
double_p x, y;
integer i, j;
```

- The usage is identical to the default data types

```
double *x, *y;
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

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.

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

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

enum weekdays { MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY };

typedef struct {
   char *name;
   char *abbr; // abbreviation
} week_day_s;

const week_day_s days_en[] = {
   [MONDAY] = { "Monday", "mon" },
   [TUESDAY] = { "Tuesday", "tue" },
   [WEDNESDAY] = { "Wednesday", "wed" },
   [THURSDAY] = { "Thursday", "thr" },
   [FRIDAY] = { "Friday", "fri" },
};
```

lec05/demo-struct.c

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

```
const week_day_s days_cs[] = {
   [MONDAY] = { "Pondeli", "po" },
   [TUESDAY] = { "Utery", "ut" },
   [WEDNESDAY] = { "Streda", "st" },
   [THURSDAY] = { "Ctvrtek", "ct" },
   [FRIDAY] = { "Patek", "pa" },
};

int main(int argc, char *argv[], char **envp)
{
   int day_of_week = argc > 1 ? atoi(argv[1]) : 1;
   if (day_of_week < 1 || day_of_week > 5) {
      fprintf(stderr, "(EE) File: '%s' Line: %d -- Given day of week out of range\n"
    , __FILE__, __LINE__);
      return 101;
   }
   day_of_week -= 1; // start from 0
```

lec05/demo-struct.c

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.

```
   _Bool cz = 0;
   while (*envp != NULL) {
      if (strstr(*envp, "LC_CTYPE") && strstr(*envp, "cs")) {
         cz = 1;
         break;
      }
      envp++;
   }
   const week_day_s *days = cz ? days_cs : days_en;

   printf("%d %s %s\n",
         day_of_week,
         days[day_of_week].name,
         days[day_of_week].abbr);
   return 0;
}
```

lec05/demo-struct.c

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

    `if ( i & 0x0010) ...`

  - Clear the 4 bit of i

    `i &= ∼0x0010;`

- We can give names to particular bits

```
#define RED   1
#define GREEN 2
#define BLUE  3

i |= RED;          // sets the RED bit
i &= ~GREEN;       // clears the GREEN bit
if (i & BLUE)  ...  // test BLUE bit
```


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!
```

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

```c
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));
```

```c
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

```c
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

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
    Like a local static variable
  - 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`

```c
#define GRAPH_SIZE 1000

typedef struct {
   ...
} edget_s;

typedef struct {
   edges_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`

```c
#include "graph.h"

graph_s graph_global = { NULL, GRAPH_SIZE };

graph_s* load_graph(const char *filename)
{
   ...
}
```

- `main.c`

```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

```c
#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.

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

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

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`*lib*, 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.

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.

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

OBJS=$(patsubst %.c,%.o,$(wildcard *.c))

TARGET=program

bin: $(TARGET)

$(OBJS): %.o: %.c
  $(CC) -c $< $(CFLAGS) $(CPPFLAGS) -o $@

$(TARGET): $(OBJS)
  $(CC) $(OBJS) $(LDFLAGS) -o $@

clean:
  $(RM) $(OBJS) $(TARGET)
```

ccache
CC=clang make vs CC=gcc make

- **The order of the files is important during the linking!**

Part III

Part 3 – Assignment HW 05

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

Eventually with dynamic allocation and structures

- **Assignment**:

`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**: 04.04.2020, 23:59:59 PDT, Bonus part 16.05.2020

PDT – Pacific Daylight Time

Summary of the Lecture

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