Einführung in C - Introduction to C

6. Advanced data types

Prof. Dr. Eckhard Kruse

DHBW Mannheim

Structure

Definition

A **Structure** is a composition of variables of (potentially different) types.

- The keyword **`struct`** defines the structure, variables are then declared using the structure name (or directly after the closing brace of the structure definition).
- Structures may be nested with other complex types such as arrays or other structures.
- Structure members are accessed via the . operator
- Structure variables can be initialized using curly braces.

struct struct_name { datatype struct_member; … } struct_variables;

```
struct datum
{
   unsigned char tag;
   unsigned char monat;
   int jahr;
};

struct person
{
   char name[50];
   char vorname[50];
   struct datum geboren;
};
```

```
struct person
   dozent, studierende[100];

struct datum weihnachten={ 24, 12,
2021 };
…
strcpy(dozent.name, "Kruse");
dozent.geboren.tag = 12;
…
```

(C99 designated initializers:

```
struct datum weihnachten={ .tag=24,
.monat=12, ... };                    )
```

Enum

Definition

The **enum** type specifier is short for "enumerated data". The user can define a fixed set of words that a variable of type enum can take as its value. The words are assigned integer values either automatically by the compiler or explicitly in the code.

```
enum direction
{
  north,
  east,
  south,
  west
};

enum direction my_direction;
my_direction = west;
```

```
enum processState {
     init=1, running=2,
     suspended=3, error=-1
};

enum processState state;

state = init;
```

Address list

Exercise

address_list.c

Code snippet 601

Union

Definition

A **union** is like a structure in which all of the members are stored at the same address. Only one member can be in a union at one time.

```
union flint {
   float u_f;
   int u_i;
};
union flint var;
...

var.u_f = 23.5;
printf("value is %f\n", var.u_f);
var.u_i = 5;
printf("value is %d\n", var.u_i);
```

```
struct my_safe_union {
      int type_in_union;
      union {
               float   un_float;
               char    un_char;
               int     un_int;
      } vt_un;
};

struct my_safe_union un;
...
if(un.type_in_union==1)
   un.vt_un.un_int=...
```

Why should we want this?

- save memory
- sort of 'low-tech polymorphism' (= same data object can be of different forms)
- risk of making errors (reading the wrong type) → thus, see right example
- used only rarely in real-life C

Bitfield

Definition

A **Bitfield** is a special struct which provides information about how many bits are used for each member.

```
struct state {
    unsigned int mode0to3 : 2;
    unsigned int enabled : 1;
    unsigned int running : 1;
};

struct state sys_state;

sys_state.enabled = 1;
sys_state.running = 1;
sys_state.mode0to3 = 0;
```

```
#define Mode0to3    (1|2)
#define Enabled      4
#define Running      8

unsigned char sys_state;

sys_state= Enabled | Running;

// read mode
currentmode = sys_state & Mode0to3;
```

- very rarely used
- typically preferred: direct bit manipulation using | & operators and constants for the bits (see right side) → full control about which bit is stored where.

Typedef

Definition

The **typedef** command is used to define own data types based on existing data types:

typedef existing_type new_type;

```
typedef int my_number_type;
…

my_number_type var1, var2, var3;

var1 = 10;
var2 = 20;
var3 = 30;
```

```
typedef struct {
  int quot; /* quotient */
  int rem; /* remainder */
} div_t;

…
div_t result;
result.quot=a/b;
result.rem=a%b;
```

- Can be used to facilitate later changes and to improve understandability (only one line to change a data type throughout the program)
- Usually not used for basic data types
- Most often used with structs (to shorten the declaration of struct variables)

Data type conversion

Definition

The **cast operator** converts the type of a variable or expression into another type by writing the target type in round brackets (= the cast) in front of it.

(new_type)operand_of_some_other_type

```
int i;
float f;
long l;
…
i = f;        // implicit casting+compiler warning (precision loss)
i = (int)f;   // no warning
l = i;        // implicit casting without warning
```

- Casts can only be used for 'simple/straighforward' type conversions.
  - Converting e.g. a string containing a number into an integer or float type requires to call a conversion function from the standard libraries.
- Some casting is done implicitly by the compiler (some with a warning).
- Casting is especially important when working with pointers.

Constants

Definition

Putting the keyword **const** in front of a data type, means that the value is constant, i.e. it cannot be reassigned throughout the program.

```
const double pi = 3.14159265;
const int disabled=0;
const int enable=1;
const int error=255;

const int prime[]={ 2, 3, 5, 7, 11 };

pi=4.0; // -> compiler error
```

Alternative:

```
enum state {
     disabled=0,
     enabled=1,
     error=255
};
```

- A const must be assigned a value when it is declared.
- Also the #define directive can be used to define constants, but const has a significant advantage. Which one?
- An alternative to *const int* is to use enum *(see right side above)*

Typedef + const

Exercise

typedef + const

Have a look at your previous programs. Can you improve them using the keywords from the previous slides?

- Use typedef to introduce specific types that are meaningful und important for your program (especially when working with structures)
- Check whether variables (or constants defined with #define) can be defined as const. What happens, if you assign a new value to a constant?

602