Structures

Dr. Donald Davendra *Ph.D.*

Department of Computing Science, FEI VSB-TU Ostrava

Derived and Structured Data Types

basic data type - part of the standard language,
preprocessor - without parameters, usees macros rather choose constants
derived data types

```
typedef <type definition> <identifier>;
```

own definition of the data type of structured data types
struct and **union**
and their own structure definition.

Derived and structred data types

declaration	importance
`typ name;`	`typ`,
`typ name[];`	array of `typ`,
`typ name[3];`	array (fixed size) of three items of type `typ` (`name[0]`, `name[1]`, `name[2]`),
`typ *name;`	pointer of `typ`,
`typ *name[];`	(open) array of pointers to type `typ`,
`typ *(name[]);`	(open) array of pointers to type `typ`,
`typ (*name)[];`	a pointer to the (open) type field type `typ`,
`typ name();`	function returning a value of type `typ`,
`typ *name();`	function returns a pointer to a value of type `typ`,
`typ *(name());`	function returns a pointer to a value of type `typ`,
`typ (*name)();`	pointer to function returning `typ`.

General procedure - from the inside out. The procedure for the correct interpretation of the definition:

1. start with the right ID and look for round or square brackets (if any);
2. interpret these two brackets on the left and look for the asterisk;
3. if we encounter a right parenthesis (at any level of nesting), let us return and apply the rules one and two for everything between the parentheses;
4. apply the type specification.

Example of a more complex type design

```
char *( *( *var) ()) [10];
  7   6  4  2 1    3    5
```

1. identifier is declared as var
2. pointer
3. function, which returns
4. pointer
5. array of ten elements, which are
6. pointers to
7. values of type char.

Example of a more complex type design 2

```
unsigned int *(*const *name[5][10]) (void);
```

The identifier name is a two-dimensional array by a total of fifty elements. The elements of this array are pointers to the parameters that are constant. These constant indicators show the type of function that has pointer arguments and returns a value of type unsigned int.

```
double (*var (double (*)[3])) [3]
```

The function returns a pointer to an array of three values of type double. The argument, as well as the return value is a pointer to an array of three elements of type double.

The argument of the function is a construct called abstract declaration. Generally it is a declaration without an identifier. To simplify and clarify the use of abstract declarations use typedef structure.

declaration	importance
`int *`	pointer to type `int`,
`int *[3]`	array of three pointers to `int`,
`int (*)[5]`	pointer to an array of five elements of type `int`
`int *()`	fce without specifying arg. returning uk. `int`,
`int (*) (void)`	uk. the destitute fci arguments returning `int`,
`int (*const [])`	uk. for an unspecified number of constant passed to the function
`(unsigned int, ...)`	each of which has a first argument in `unsigned int` and an unspecified number of additional arguments.

Allows you to define an enum constants.
`enum [tag] enum-list [declarator];`

1. `enum` is a keyword initiating definition of values enumerated type;
2. `tag` is optional "tag", used mainly in the style of K & R C, with a structure typedef frequency of its use is declining;
3. `enum-list` is a list of constants enumerated type with possible explicitly assigned value, see example below, otherwise it becomes the first enum constant value of zero, the second value one,... Each successor has a value one greater than its predecessor;
4. `declarator` is optional list of variables of type `enum`.

enumerator

```
typedef enum {
     Back = 8, Tab = 9, Esc = 27, Enter = 13,
     Down = 0x0150, Left = 0x014b, Right = 0x014d,
     Up = 0x0148, NUL = 0x0103, Shift_Tab = 0x010f,
     Del = 0x0153, End = 0x014f, Home = 0x0147,
     Ins = 0x0152, PgDn = 0x0151, PgUp = 0x0149
} key_t;
int key;
  ...
  else if ((key == Left) || (key == Back))
  ...
  else if (key == Enter)
  ...
  else if (key == Esc)
  ...
  else if ...
```

Structs

```
struct [<struct-type-name>] {
   [<type> <variable-name[, variable-name, ...]>] ;
   [<type> <variable-name[, variable-name, ...]>] ;
   ...
} [<structure variables>] ;
```

. dot operator - selector of the memebre of the struct (static)

-> - selector of struct members using pointer declarations

Struct example

```
typedef
 struct {float re, im;} complex;

 complex cislo,
         im_jednotka = {0, 1};
 cislo.re = 12.3456;
 cislo.im = -987.654;
 printf("re = %10.5f im = %10.5f\n",
         im_jednotka.re, im_jednotka.im);
 printf("re = %10.5f im = %10.5f\n",
         cislo.re, cislo.im);
```

```
typedef
  struct {int ev_cislo;
          char nazev[ZNAKU_NAZEV + 1];
          int  na_sklade;
          float cena;
         } vyrobek;
typedef vyrobek zbozi[POLOZEK_ZBOZI];

 vyrobek a = {8765, "nazev zbozi na sklade",
              100, 123.99};
 vyrobek *ppolozky;
 ...
 ppolozky->ev_cislo = 1;
```

```
polozky[0].ev_cislo = 0;
strcpy(polozky[0].nazev, "polozka cislo 0");
polozky[0].na_sklade = 20;
polozky[0].cena = 45.15;

ppolozky = polozky + 1;
ppolozky->ev_cislo = 1; /* (*ppolozky).ev_cislo = 1;*/
strcpy(ppolozky->nazev, "polozka cislo 1");
ppolozky->na_sklade = 123;
ppolozky->cena = 9945.15;

printf(FORMAT_VYROBEK, a.ev_cislo, a.na_sklade, a.cena, a.nazev);
printf(FORMAT_VYROBEK, polozky[0].ev_cislo, polozky[0].na_sklade,
       polozky[0].cena, polozky[0].nazev);
printf(FORMAT_VYROBEK, ppolozky->ev_cislo, ppolozky->na_sklade,
       ppolozky->cena, ppolozky->nazev);
```

```
re =     0.00000 im =     1.00000
re =    12.34560 im = -987.65399
cislo: 8765 pocet: 100 cena:   123.99 nazev:nazev zbozi na sklade
cislo:    0 pocet:  20 cena:    45.15 nazev:polozka cislo 0
cislo:    1 pocet: 123 cena:  9945.15 nazev:polozka cislo 1
```

Structure of FILE

Type FILE, defined in stdio.h header file is:

```
typedef struct {
    short           level;
    unsigned        flags;
    char            fd;
    unsigned char   hold;
    short           bsize;
    unsigned char *buffer, *curp;
    unsigned        istemp;
    short           token;
} FILE;
```

Incomplete structures declaration

```
struct A;                          /* neuplna */
struct B {struct A *pa};
struct A {struct B *pb};
```

the solution is made possible by the pointer whose its size is known.

```
union [<union type name>] {
  <type> <variable names> ;
   ...
} [<union variables>] ;
```

Syntax as a **struct**.
Semantics (!) - Union of the items can be used at any one time only once.
Implementation: memory space, reserved for the union is so large to accommodate a single (largest memory item).

It is up to the programmer who is working with the union element to ensure that it is designed properly.

Bit Field

Bit field is an integer, placed in the specified number of bits. These bits form a contiguous area of memory. Bit field can contain multiple integer entries. We can create a bit array of three classes:

1. free bit field
2. signed bit field,
3. unsigned bit field.

Bit fields can be declared only as members of the struct or union. An expression that we write for an item identifier and the colon represents the field size in bits.

Struktura ftime detailně.

```
struct ftime {
    unsigned ft_tsec  : 5;  /* Two seconds */
    unsigned ft_min   : 6;  /* Minutes     */
    unsigned ft_hour  : 5;  /* Hours       */
    unsigned ft_day   : 5;  /* Days        */
    unsigned ft_month : 4;  /* Months      */
    unsigned ft_year  : 7;  /* Year - 1980 */
};
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ft_hour					ft_min						ft_sec				
hodiny					minuty						sekundy/2				

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ft_year							ft_month				ft_day				
rok-1980							měsíc				den				