

Computer Programming

User-defined types

Marius Minea

marius@cs.upt.ro

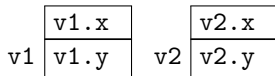
4 December 2014

Structures are for compound values

group elements of potentially different types (logically connected)
can use/assign/pass/return entire aggregate value, or parts of it

```
struct len { // type is 'struct len', 'len' = structure tag
    double val;
    char unit[3];
};
struct len d1 = { 60, "km" }; // declaration + initialization
```

```
struct vect { // type: 'struct vect'
    double x, y;
} v1, v2; // two vars of this type
```



Structure elements are called *fields*

of any type, but *NOT* the *same* structure type (NOT recursive)

Using fields: *var_name.field_name*

the dot `.` is the postfix *selection operator*

```
struct vect p1; p1.x=2; p1.y=3; printf("%f %f\n", p1.x, p1.y);
```

Allowed operations

Structures may be *assigned*: `struct vect v1={2, 3}, v2; v2=v1;`

We may write *compound structure values*: use (type cast)

```
struct vect v1; v1 = (struct vect){-4, 5};
```

Structures may be *passed* to and *returned* from functions
for large structures should pass/return pointers (less copying)

```
struct vect add(struct vect v1, struct vect v2) {  
    return (struct vect){ v1.x + v2.x, v1.y + v2.y };  
}
```

We *may NOT compare* structures with logical operators (`==`, `!=`)
⇒ must compare field by field: `if (v1.x==v2.x && v1.y==v2.y)...`

Reason: *alignment* in memory may cause spaces between fields
value of these hidden bytes is undetermined ⇒ don't use `memcmp`

Structures and arrays

In C, aggregated (compound) types may be combined arbitrarily
arrays of structures, structures with array or structure fields, etc.

Define types to *logically group data*

E.g. replace two related arrays of same range by array of structures:

```
char* name_mo[12] = { "January", /* ... , */ "December" };
char day_mo[12] = { 31, 28, 31, 30, /* ... , */ 30, 31 };
// better:
struct month {
    char *name; // pointer to string constant
    int days;
};
struct month mo[12] = {{"January",31}, ..., {"December",31}};
```

Structures and typedef

typedef allows us to give new names to existing types

General form: `typedef existing-type new-type-name;`

(like variable declaration + `typedef` in front \Rightarrow names a *type*)

e.g. `typedef double real; typedef struct vect vect_t;`
`typedef int (*cmpfun)(const void *, const void *);`

We can give the name directly in the type definition

```
typedef struct student { /*some fields */} student_t;
```

may omit structure tag (after **struct**) and use just new name

```
typedef struct { /*some fields */} student_t;
```

or separately define synonym and structure type (in either order)

```
struct student { /*some fields */}; //defines type
```

```
typedef struct student student_t; //defines synonym
```

Structures and strings

```
typedef struct {  
    char name[64];    // fixed-length array  
    char *addr;      // only ADDRESS, NO memory for chars  
} student_t;        // declares name for structure type  
student_t s;
```

s.name is *array*: we can copy or read a string:

CANNOT assign ~~s.name =~~, it's a CONSTANT address!

```
strcpy(s.name, "Stefanovici"); //NOT s.name = ...
```

```
if (scanf("%63s", s.name) == 1) ...
```

s.addr is *pointer*: we must assign a *valid* address

e.g., a string constant: s.addr = "str. Linistei 2";

or dynamically allocated memory:

```
if (fgets(buf, sizeof(buf), stdin) s.addr = strdup(buf);
```

Field names are only visible *inside* the structure

⇒ cannot use fieldname by itself, only *varname.field*

⇒ different structure types can have fields with same name

Pointers to structures

Like any variable, a structure can be accessed through a pointer:

```
struct student s, *p = &s; (*p).final_grade = 9.50;
```

The `->` operator is shorthand for indirection followed by selection:

use: `pointer->fieldname` means: `(*pointer).fieldname`

Operators `.` and `->` have the *highest precedence*, like `()` and `[]`

<code>p->x++</code>	means	<code>(p->x)++</code>	<code>-></code> has priority
<code>++p->x</code>	means	<code>++(p->x)</code>	<code>-></code> has priority
<code>*p->x</code>	means	<code>*(p->x)</code>	<code>-></code> has priority
<code>*p->s++</code>	means	<code>*((p->s)++)</code>	<code>++</code> has priority over <code>*</code>

Recursive data structures

A structure field may not be a structure of the *same type*
size of the structure would be undefined/infinite

But can have *address* of the same type of structure (a pointer)
⇒ *recursive, linked* datastructures (lists, trees, etc.)

```
struct wl {           // struct wl incompletely defined type at this
    char *word;       // word: the actual data
    struct wl *next;  // pointer to same type of structure
};                    // type definition is now complete
```

Binary tree with integer nodes

```
typedef struct t tree_t; // tree_t is name for incomplete type
struct t {
    int val;
    tree_t *left, *right; // use typedef name
};                        // type struct t now complete, same as tree_t
```


Structures with bitfields

We usually want to represent information as compactly as possible but don't use too restrictive assumptions! (see Y2K problem)

date = 32-bit int: sec, min (0-59): 6 bits, hour (0-23), day (1-31): 5 bits, month (1-12): 4 bits, year (1970 + 0-63): 6 bits

```
struct date { // structure with bitfields
    unsigned sec : 6, min : 6; // 6 indicates bit count
    unsigned hour : 5, day : 5; // each field must have width
    unsigned month: 4; // use only integer types
    unsigned year: 6;
} data = {0, 0, 17, 19, 5, 39 }; // 17:00:00, 19.05.(1970+39)
```

We can directly write:

```
printf("%u.%u\n", data.day, data.month);
```

Nameless fields can control space used: `int: 2; //2 bits`
or force storing data starting in the next byte `int: 0;`

Structures and alignment

Compiler *aligns* each data type in memory for best processor access
can find out with `_Alignof` operator

```
printf("%zu %zu\n", _Alignof(int), _Alignof(char*)); //4 8
```

Structure fields are in order but need not be in consecutive bytes
`offsetof(structuretype, fieldname)` tells where (from `stddef.h`)

```
typedef struct { char s[3]; char val[8]; } s1_t;
typedef struct { char s[3]; double val; } s2_t;
printf("%zu %zu\n", offsetof(s1_t, val), sizeof(s1_t)); // 3 11
printf("%zu %zu\n", offsetof(s2_t, val), sizeof(s2_t)); // 8 16
// because _Alignof(double) is 8 bytes
```

Always check if your code relies on a specific alignment!
(e.g., to directly read an entire structure value from a file)

Structures with flexible array members

Sometimes the size of an array field is not known statically

⇒ *last* member of a structure may be an incompletely defined array

```
typedef struct {  
    char *fname;  
    unsigned argc; // number of args  
    int args[]; // default length is zero  
} func_t; // type for a function of integers
```

When declaring `func_t f`; the array has length 0 (no elements)

But, can dynamically create a structure of the desired size:

```
func_t *fp = malloc(sizeof(func_t) + n * sizeof(int));  
// or: ... + sizeof(int [n])  
if (fp) {  
    fp->argc = n;  
    for (int i = 0; i < n; ++i)  
        fp->args[i] = ...  
}
```

Enumeration type

gives *names* to integer values (constants)

⇒ use when names are more suggestive than integers

```
enum univ_mo {jan=1, feb, mar, apr, may, jun, oct=10, nov, dec};
```

defines type `enum univ_mo` (the keyword is part of the type name)

Default: increasing sequence of values, starting at 0

Can explicitly specify values (restarts count); values may repeat

An enumeration type is an *integer* type ⇒ values used as ints

```
enum {Su, M, Tu, W, Th, F, Sa} day_t; // anonymous type
int work_hours[7]; // per weekday
for (int day = M; day <= F; ++day) work_hours[day] = 8;
```

Enumeration constants are used by themselves (one namespace)

⇒ A constant name may *NOT* be used in distinct enumerations

Unions

Used to store a value which may have one of several *different* types

Syntax: as for structures, but with keyword **union**

List of fields is a *list of variants*

a structure contains *all* declared fields

a union contains *exactly one* variant; has size of *largest* type

```
struct {
    enum { INT, DBL, STR } type; // remembers which variant
    union { // anonymous union type
        int i;
        double r;
        char *s;
    } u;
} v; // three variants for a value
char s[32]; if (scanf("%31s", s) == 1) {
    if (isdigit(*s)) // starts with digit or contains dot
        if (strchr(s, '.')) { v.type=DBL; sscanf(s, "%lf", &v.u.r); }
        else { v.type = INT; sscanf(s, "%d", &v.u.i); }
    else { v.type = STR; v.u.s = strdup(s); }
}
```