Computer Programming

User-defined types

Marius Minea

marius@cs.upt.ro

5 December 2016

Structures are for compound values

group (logically connected) elements of potentially *different types*

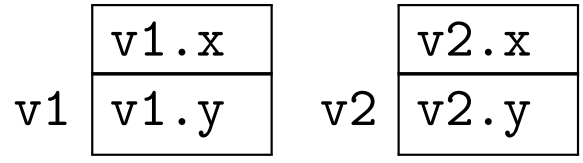
can use/assign/pass/return *entire aggregate* value, or *parts* of it
  structures are *first-class* values in C

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

Structures correspond to *product types*
  set of possible values is *cartesian product* of component types
above: *any* real number with *any* 3-char string

Structures have named *fields*

```
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 (~~infinite recursion~~)

Access fields as: *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);
```

Can write *structure values*, with or without field names:

```
struct vect v1 = { 2, 3 }, v2 = { .x = 4, .y = 5 };
```

Operations with structures

We may *assign* structures: `struct vect v1={2, 3}, v2; v2=v1;`

Except for initialization, need (type cast) for aggregate values:

```
struct vect v3, v4;
v3 = (struct vect){-4, 5};
v4 = (struct vect){ .x = -1, .y = 2};
```

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 (==, !=)
$\Rightarrow$ 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 hidden bytes is undetermined $\Rightarrow$ also 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 ⇒ names a *type*)

e.g. `typedef double real;` `typedef struct vect vect_t;`

```
typedef int (*cmpfun_t)(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
$\Rightarrow$ cannot use fieldname by itself, only *varname*.*field*
$\Rightarrow$ different structure types can have fields with same name

Pointers to structures and the -> operator

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`

Use *pointers* to large structures as function arguments:
avoids needless copying of data onto stack

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

`p->x++`	means	`(p->x)++`	-> has priority
`++p->x`	means	`++(p->x)`	-> has priority
`*p->x`	means	`*(p->x)`	-> has priority
`*p->s++`	means	`*((p->s)++)`	first ++ then * (right assoc.)

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)
$\Rightarrow$ *recursive, linked* datastructures (lists, trees, etc.)

List of words:

```
struct wl {       // struct wl: incompletely defined type
  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 want compact, efficient representations
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
```

If you define structures for easier work with certain file formats
check that offsets are the same as in the file (no unused bytes)

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
```

Declaring `func_t f;` is useless, array has length 0 (no elements)

⇒ cannot initialize statically, cannot pass struct as argument

But, can dynamically create a structure of the desired size:
and pass *pointer* to struct as function argument

```
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 and union types

two other kinds of user-defined types

declaration *syntax*: with keyword + tag + braces
(similar to structures)

enumeration: just named integer values

union: declares a type which is the union of several types
may contain one value of any of the types

Enumeration type

gives *names* to integer values (constants)
⇒ use for *readability* (names are more suggestive than ints)

```
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
  set union between type values, also called *sum type*

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

```
union int_or_float {
  uint32_t u;
  float f;
} v;    // a variable v of this union type
```

can store *either* an int in `v.u` *or* a float in *v.f*
must remember which (can't tell from value, either option is valid)

```
v.f = .5;
printf("%x\n", v.u); // 3F000000: binary rep of float 0.5
```

Use unions with enums

Use a structure type with:

- a **union** for the actual value
- an **enum** to tell which kind of value it is

```
struct ids {
  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 = (struct ids){ .type = STR, .u.s = strdup(s) };
}
```