Chapter Eight

Structures

Lecture slides prepared by Alistair Moffat

Chapter 8

- Data abstraction.
- Structures and structure operations.
- Structures and functions.

8.1 Declaring structures

A structure is a collection of individual variables of possibly different types, accessed via component names.

```
#define PLANETSTRLEN 20

typedef char pstr_t[PLANETSTRLEN+1];

typedef struct {
    pstr_t  name, orbits;
    double  distance;      /* million km */
    double  mass;          /* kilograms */
    double  radius;        /* kilometers */
} planet_t;

planet_t one_planet;
```

8.1 Declaring structures

The six components can be used as individual variables, as if they had been separately declared:

```
strcpy(one_planet.name, "Earth");
strcpy(one_planet.orbits, "Sun");
one_planet.distance = 149.6;
one_planet.mass = 5.976e+24;
one_planet.radius = 6378.1;
```

8.1 Declaring structures

Structures are normally set up using `typedef`, and then those types get used in the remainder of the program:

- `struct.c`

Note the array-like initialization.

8.2 Operations on structures

Structures of the same type can be assigned, even if they contain arrays.

The complete contents of the RHS structure variable – including any array components – are copied.

After the assignment, all components of the two structures have identical values.

8.2 Operations on structures

It is not possible for two structures to be compared for equality, or relative ordering.

You always need to write your own `cmp(p1, p2)` function that examines the fields that you are interested in. Write it so that if `p1` come before `p2` you return a –ve value, if `p1` is equal to `p2` you return 0, and if `p1` comes after `p2` you return a +ve value. Both `p1` and `p2` will normally be pointers.

Structures are read and written one component at a time, using the appropriate format descriptor.

8.2 Operations on structures

Structure hierarchies are used to show relationships between data elements.

Common elements are abstracted into separate declared types. Like components in different structures should be given the same names.

- `nested.c`

A consistent naming strategy, such `_t`, helps avoid confusion between types and variables.

8.2 Operations on structures

All of these variables are declared:

```
jane                          staff_t
jane.name                     fullname_t
jane.datecommenced.mm         int
jane.annualsalary             int
bill                          student_t
bill.dob                      date_t
bill.dob.mm                   int
bill.name.given               char[41]
bill.name.given[3]            char
bill.subjects                 subject_t[8]
bill.subjects[1].enrolled     date_t
bill.subjects[1].enrolled.yy  int
bill.subjects[1].finalmark    int
```

8.2 Exercise 1

Define a structure to account for this situation:

Cars have six-character registration numbers, and two dates associated with them – the date the car was first registered, and the date that the current registration expires. Each car also has fields (40-byte strings) for manufacturer, make, body type, and color; and a field to record the number of owners it has had.

8.3 Structures, pointers, and functions

Structures are passed into functions by making a copy of the argument into a local argument variable, in the same way as scalar variables.

Changes made to the argument variable are discarded when the function returns.

```
planet_t planet;
print_planet(planet);
```

8.3 Structures, pointers, and functions

Functions can return structures. The value to be returned is composed in a local variable, and then assigned to a different variable in the calling function:

```
planet_t planet;
planet = read_planet();
```

In these two respects, structures and arrays differ markedly.

8.3 Structures, pointers, and functions

The address of a structure can stored in a pointer variable of the correct type:

```
planet_t *p;
planet_t planet;
p = &planet;
```

C provides a shorthand operator to assist: `p->mass` is the same as `(*p).mass`.

Still need to use one set of parentheses when reading: `&(p->mass)` and `&(p->distance)`.

8.3 Structures, pointers, and functions

It is usual to pass a structure pointer to a function, rather than a structure. Doing so avoids the cost of copying, and allows components to be changed.

Modification of a structure via a pointer argument also allows the function to return a flag.

There are no structure expressions, so requiring that a structure variable always underpin the argument is not restrictive in any way.

8.4 Structures and arrays

Structures can be used as the base type of an array.

```
#define MAXBODIES 100

int nplanets=0;
planet_t planets[MAXBODIES];
```

This allows `planets[i].distance` and so on.

Pointer arithmetic works correctly: `planets+i` is a pointer to the `i`'th element of `planets`.

8.4 Structures and arrays

An array and its buddy variable `nplanets` can be combined into a new structure, so that they stay together:

```
typedef struct {
    int nplanets;
    planet_t planets[MAXBODIES];
} solar_system_t;

solar_system_t solar_system;
```

Now use `solar_system.planets[i].mass` to access one field.

A complete solar system can be passed to a function as a single argument. Wow!

8.4 Exercise 2

Write a function `bigger_planet(solar_system_t *S, int p1, int p2)` that returns 1 if the `p1`'th planet in the solar system described by `*S` is heavier than the `p2`'th one; returns 0 if it is smaller; and returns −1 if either of the planet indices is invalid.

Chapter 8 – Summary

- Structures provide hierarchical data abstraction in the same way that functions provide control abstraction.
- Structures can be assigned, and can be passed in to and returned from functions (but it is more usual to pass a structure pointer).
- A single structure variable might be a quite complex package of related information, all traveling to the same place at the same time.