```
#define NAMESTRLEN 40
#define MAXSUBJECTS 8

typedef char namestr[NAMESTRLEN+1];

typedef struct {
    namestr     given, others, family;
} fullname_t;

typedef struct {
    int         yy, mm, dd;
} date_t;

typedef struct {
    int         subjectcode;
    date_t      enrolled;
    int         status;
    int         finalmark;
} subject_t;

typedef struct {
    fullname_t  name;
    int         employeenumber;
    date_t      dob;
    date_t      datecommenced;
    int         status;
    int         annualsalary;
} staff_t;

typedef struct {
    fullname_t  name;
    int         studentnumber;
    date_t      dob;
    int         nsubjects;
    subject_t   subjects[MAXSUBJECTS];
} student_t;

staff_t     jane;
student_t   bill;
```

Figure 8.4: Declaration of nested structures.

Note how the common elements have been abstracted out into separate declared types: `fullname_t`, `date_t`, and so on. Components in different structures can have the same name – for example, the components `name` and `dob` are common to both `staff_t` and `student_t`. Indeed, when components in two different structures have the same interpretation, it makes perfect sense for them to be named alike.

Note also that this is just a sketch – in a real personnel or student record management system there would be literally dozens of other types and fields involved (address, entrance marks, payroll history, annual leave records, next of kin, tax paid, applications for special consideration, enrollment history for previous years of study, and so on) and the structures would be correspondingly more complex. But the same

overall rules apply – *data abstraction* is used to create a hierarchical structure to the information that must be maintained, in the same way that function abstraction allows grouping of repeated execution patterns on that data.

Finally in connection with Figure 8.4, note the use of the "`_t`" convention on type names. There are other possible conventions, including the use of initial uppercase letters for types, `Fullname`, `Date`, `Subject`, and so on. In some languages this latter approach is mandatory. There is no requirement in C that types be differentiated in any particular way; nevertheless, without some kind of convention like this, you are quickly going to lose track of which identifiers are being used for types and which are being used for variables. In this book the "`_t`" suffix is used to distinguish types from variables and functions. Use of this rule (or a similar style) allows declarations of the form `date_t date`, which is probably more helpful than the alternative of using `typedef` to create a type `date`, and then hunting for an alternative name for the actual variable being declared: `date dte`, for example.

> Type names for structures and other types can be any valid identifier. But for readability, and ease of maintenance, you should adopt a sensible convention and use it methodically.

Another way of thinking about structures, and nested structures, is to imagine that you are going on a skiing holiday. Your toothbrush, toothpaste, and other toiletries get packed into a small bag, and then it gets zipped closed. They correspond to one sub-structure. Your passport, travelers checks, and credit cards then get sealed into a document wallet – another sub-structure. Perhaps some underclothes get put into yet another package; your gloves, goggles, and hat into another; and some pairs of shoes into yet another special purpose bag. Finally, all of these components, plus some shirts and trousers and jackets, get assembled together, put into a suitcase, and the suitcase closed – the main structure. Then, when you check-in at the airport for your flight, and they ask "just one bag?", you answer "yes", because by now you do have just one bag, and all the other components are inside it at the first level, or at the second level, or perhaps even at a third level. In the same way, structures allow us to manipulate a suitcase full of variables without having to list every object individually. In particular, we are able to take structures into and out of functions – the equivalent of taking a suitcase on holiday, and then bringing it back home again.

8.3 Structures, pointers, and functions

Structures are passed into functions in exactly the same manner as scalar variables of type `int` or `float` – the value of the argument expression is copied into a local argument variable of the corresponding type, and within the function the local variable is manipulated. Function `print_planet` in Figure 8.5 shows this mode of argument passing, where the types are as declared in Figure 8.1. Within the function, variable `one_planet` is local, and changes made to it are not reflected in the passed variable `planet` declared within the scope of function `main`. In the case of `print_planet` (the body of which consists of a `printf` statement that appears in Figure 8.2), this is not a problem.