# Chapter 9

# Structures and Unions

C is an easily extensible language. It can be extended by providing macros that are stored in header files and by providing functions that are stored in libraries. It can also be extended by defining data types that are constructed from the fundamental types. An array type is an example of this; it is a derived type that is used to represent homogeneous data. In contrast, the structure type is used to represent heterogeneous data. A structure has components, called *members*, that are individually named. Because the members of a structure can be of various types, the programmer can create aggregates of data that are suitable for a particular application.

## 9.1 Structures

The structure mechanism provides a means to aggregate variables of different types. As a simple example, let us define a structure that describes a playing card. The spots on a card that represent its numeric value are called “pips.” A playing card such as the three of spades has a pip value, 3, and a suit value, spades. We can declare the structure type

```
struct card {
  int    pips;
  char   suit;
};
```

to capture the information needed to represent a playing card. In this declaration, struct is a keyword, card is the structure tag name, and the variables pips and suit are members of the structure. The variable pips will take values from 1 to 13, representing ace to king; the variable suit will take values from 'c', 'd', 'h', and 's', representing the suits clubs, diamonds, hearts, and spades, respectively.

This declaration creates the derived data type struct card. It is an example of a user-defined type. The declaration can be thought of as a template; it creates the type struct card, but no storage is allocated. The tag name, along with the keyword struct, can now be used to declare variables of this type.

```
struct card   c1, c2;
```

This declaration allocates storage for the identifiers c1 and c2, which are of type struct card. An alternative scheme is to write

```
struct card {
  int    pips;
  char   suit;
} c1, c2;
```

which defines the type struct card and declares c1 and c2 to be of this type, all at the same time.

To access the members of a structure, we use the member access operator ".". Let us assign to c1 the values representing the three of spades.

```
c1.pips = 3;
c1.suit = 's';
```

A construct of the form

*structure_variable . member_name*

is used as a variable in the same way that a simple variable or an element of an array is used. If we want c2 to represent the same playing card as c1, then we can write

```
c2 = c1;
```

This causes each member of c2 to be assigned the value of the corresponding member of c1.

Programmers commonly use the typedef mechanism when using structure types. An example of this is

```
typedef   struct card   card;
```

Now, if we want more variables to represent playing cards, we can write

```
card   c3, c4, c5;
```

Note that in the type definition the identifier card is used twice. In C, the name space for tags is separate from that of other identifiers. Thus, the type definition for card is appropriate.

Within a given structure, the member names must be unique. However, members in different structures are allowed to have the same name. This does not create confusion because a member is always accessed through a structure identifier or expression. Consider the following code:

```
struct fruit {
  char   *name;
  int    calories;
};

struct vegetable {
  char   *name;
  int    calories;
};

struct fruit       a;
struct vegetable   b;
```

Having made these declarations, it is clear that we can access a.calories and b.calories without ambiguity.

Structures can be complicated. They can contain members that are themselves arrays or structures. Also, we can have arrays of structures. Before presenting some examples, let us give the syntax of a structure declaration:

*structure_declaration* ::= *struct_specifier* *declarator_list* ;
*struct_specifier* ::= struct *tag_name*
  | struct *tag_name*$_{\text{opt}}$ { { *member_declaration* }$_{1+}$ }
*tag_name* ::= *identifier*
*member_declaration* ::= *type_specifier* *declarator_list* ;
*declarator_list* ::= *declarator* { , *declarator* }$_{0+}$

An example is

```
struct card {
  int    pips;
  char   suit;
} deck[52];
```

Here, the identifier deck is declared to be an array of struct card.

If a tag name is not supplied, then the structure type cannot be used in later declarations. An example is

```
struct {
  int    day, month, year;
  char   day_name[4];          /* Mon, Tue, Wed, etc. */
  char   month_name[4];        /* Jan, Feb, Mar, etc. */
} yesterday, today, tomorrow;
```

This declares yesterday, today, and tomorrow to represent three dates. Because a tag name is not present, more variables of this type cannot be declared later. In contrast, the declaration

```
struct date {
  int    day, month, year;
  char   day_name[4];        /* Mon, Tue, Wed, etc. */
  char   month_name[4];      /* Jan, Feb, Mar, etc. */
};
```

has date as a structure tag name, but no variables are declared of this type. We think of this as a template. We can write

```
struct date   yesterday, today, tomorrow;
```

to declare variables of this type.

It is usually good programming practice to associate a tag name with a structure type. It is both convenient for further declarations and for documentation. However, when using typedef to name a structure type, the tag name may be unimportant. An example is

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

complex   a, b, c[100];
```

The type complex now serves in place of the structure type. The programmer achieves a high degree of modularity and portability by using typedef to name such derived types and by storing them in header files.

## 9.2 Accessing Members of a Structure

In this section we discuss methods for accessing members of a structure. We have already seen the use of the member access operator ".". We will give further examples of its use and introduce the member access operator –>.

Suppose we are writing a program called *class_info*, which generates information about a class of 100 students. We begin by creating a header file.