These three fields will require 16 bits, which, on a machine with 4-byte words, is a short integer. We can think of the three bit fields as follows:

Identification	Job type	Gender
bbbbbbbbb	bbbbbb	b

The following function can be used in a program designed to enter employee data into a short. The inverse problem of reading data out of the short would be accomplished with the use of masks.

```
/* Create employee data in a short int. */

short create_employee_data(int id_no, int job_type, char gender)
{
   short   employee = 0;     /* start with all bits off */

   employee |= (gender == 'm' || gender == 'M') ? 0 : 1;
   employee |= job_type << 1;
   employee |= id_no << 7;
   return employee;
}
```

Multibyte Character Constants

Multibyte characters are allowed in ANSI C. An example is 'abc'. On a machine with 4-byte words, this causes the characters 'a', 'b', and 'c' to be packed into a single word. However, the order in which they are packed is machine-dependent. Some machines put 'a' in the low-order byte; others put it in the high-order byte. (See exercise 12, on page 359.)

7.5 Enumeration Types

The keyword enum is used to declare enumeration types. It provides a means of naming a finite set, and of declaring identifiers as elements of the set. Consider, for example, the declaration

```
enum day {sun, mon, tue, wed, thu, fri, sat};
```

This creates the user-defined type enum day. The keyword enum is followed by the tag name day. The enumerators are the identifiers sun, mon, ... , sat. They are constants of type int. By default, the first one is 0, and each succeeding one has the next integer value. This declaration is an example of a type specifier, which we also think of as a *template*. No variables of type enum day have been declared yet. To do so, we can now write

```
enum day   d1, d2;
```

This declares d1 and d2 to be of type enum day. They can take on as values only the elements (enumerators) in the set. Thus,

```
d1 = fri;
```

assigns the value fri to d1, and

```
if (d1 == d2)
   .....      /* do something */
```

tests whether d1 is equal to d2. Note carefully that the type is enum day. The keyword enum by itself is not a type.

The enumerators can be initialized. Also, we can declare variables along with the template, if we wish to do so. The following is an example:

```
enum suit {clubs = 1, diamonds, hearts, spades}   a, b, c;
```

Because clubs has been initialized to 1, diamonds, hearts, and spades have the values 2, 3, and 4, respectively. In this example

```
enum suit {clubs = 1, diamonds, hearts, spades}
```

is the type specifier, and a, b, and c are variables of this type. Here is another example of initialization:

```
enum fruit {apple = 7, pear, orange = 3, lemon}   frt;
```

Because the enumerator apple has been initialized to 7, pear has value 8. Similarly, because orange has value 3, lemon has value 4. Multiple values are allowed, but the identifiers themselves must be unique.

```
enum veg {beet = 17, carrot = 17, corn = 17}   vege1, vege2;
```

The tag name need not be present. Consider, for example,

```
enum {fir, pine}   tree;
```

Because there is no tag name, no other variables of type enum {fir, pine} can be declared.

The following is the syntax for the enumeration declaration:

enum_declaration ::= *enum_type_specifier identifier* { , *identifier* }$_{0+}$;
enum_type_specifier ::= enum *e_tag* { *e_list* }
        | enum *e_tag*
        | enum { *e_list* }
e_tag ::= *identifier*
e_list ::= *enumerator* { , *enumerator* }$_{0+}$
enumerator ::= *identifier* { = *constant_integral_expression* }$_{\text{opt}}$

In general, one should treat enumerators as programmer-specified constants and use them to aid program clarity. If necessary, the underlying value of an enumerator can be obtained by using a cast. The variables and enumerators in a function must all have distinct identifiers. The tag names, however, have their own name space. This means that we can reuse a tag name as a variable or as an enumerator. The following is an example:

```
enum veg {beet, carrot, corn}   veg;
```

Although this is legal, it is not considered good programming practice.

We illustrate the use of the enumeration type by writing a function that computes the next day which uses typedef to replace the enum keyword in the type declaration.

In file next_day.c

```
/* Compute the next day. */

enum day {sun, mon, tue, wed, thu, fri, sat};

typedef   enum day   day;   /*the usual typedef trick */

day find_next_day(day d)
{
  day   next_day;

  switch (d) {
  case sun:
    next_day = mon;
    break;
  case mon:
    next_day = tue;
    break;
  case tue:
    next_day = wed;
    break;
  case wed:
    next_day = thu;
    break;
  case thu:
    next_day = fri;
    break;
  case fri:
    next_day = sat;
    break;
  case sat:
    next_day = sun;
    break;
  }
  return next_day;
}
```

Recall that only a constant integral expression can be used in a case label. Because enumerators are constants, they can be used in this context. The following is another version of this function; this version uses a cast to accomplish the same ends:

```
/* Compute the next day with a cast. */

enum day {sun, mon, tue, wed, thu, fri, sat};

typedef  enum day  day;

day find_next_day(day d)
{
  assert((int) d >= 0 && (int) d < 7)
  return ((day)(((int) d + 1) % 7));
}
```

Enumeration types can be used in ordinary expressions provided type compatibility is maintained. However, if one uses them as a form of integer type and constantly accesses their implicit representation, it is better just to use integer variables instead. The importance of enumeration types is their self-documenting character, where the enumerators are themselves mnemonic. Furthermore, enumerators force the compiler to provide programmer-defined type checking so that one does not inadvertently mix apples and diamonds.

7.6 An Example: The Game of Paper, Rock, Scissors

We will illustrate some of the concepts introduced in this chapter by writing a program to play the traditional children's game called "paper, rock, scissors." In this game each child uses her or his hand to represent one of the three objects. A flat hand held in a horizontal position represents "paper," a fist represents "rock," and two extended fingers represent "scissors." The children face each other and at the count of three display their choices. If the choices are the same, then the game is a tie. Otherwise, a win is determined by the rules:

Paper, Rock, Scissors Rules

- § Paper covers the rock.
- § Rock breaks the scissors.
- § Scissors cut the paper.

We will write this program in its own directory. The program will consist of a *.h* file and a number of *.c* files. Each of the *.c* files will include the header file at the top of the file. In the header file we put #include directives, templates for our enumeration types, type definitions, and function prototypes: