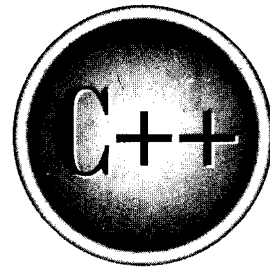

Chapter 7

Structures, Unions, Enumerations, and User-Defined Types

The C language gives you five ways to create a custom data type:

1. The *structure*, which is a grouping of variables under one name and is called an *aggregate* data type. (The terms *compound* or *conglomerate* are also commonly used.)
2. The *bit-field*, which is a variation on the structure and allows easy access to individual bits.
3. The *union*, which enables the same piece of memory to be defined as two or more different types of variables.
4. The *enumeration*, which is a list of named integer constants.
5. The **typedef** keyword, which defines a new name for an existing type.

C++ supports all of the above and adds classes, which are described in Part Two. The other methods of creating custom data types are described here.

Note *In C++, structures and unions have both object-oriented and non-object-oriented attributes. This chapter discusses only their C-like, non-object-oriented features. Their object-oriented qualities are described later in this book.*

Structures

A structure is a collection of variables referenced under one name, providing a convenient means of keeping related information together. A *structure declaration* forms a template that may be used to create structure objects (that is, instances of a structure). The variables that make up the structure are called *members*. (Structure members are also commonly referred to as *elements* or *fields*.)

Generally, all of the members of a structure are logically related. For example, the name and address information in a mailing list would normally be represented in a structure. The following code fragment shows how to declare a structure that defines the name and address fields. The keyword **struct** tells the compiler that a structure is being declared.

```
struct addr
{
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
};
```

Notice that the declaration is terminated by a semicolon. This is because a structure declaration is a statement. The type name of the structure is **addr**. As such, **addr** identifies this particular data structure and is its type specifier.

At this point, *no variable has actually been created*. Only the form of the data has been defined. When you define a structure, you are defining a compound variable type, not a variable. Not until you declare a variable of that type does one actually exist. In C, to declare a variable (i.e., a physical object) of type **addr**, write

```
struct addr addr_info;
```

This declares a variable of type **addr** called **addr_info**. In C++, you may use this shorter form.

```
addr addr_info;
```

As you can see, the keyword **struct** is not needed. In C++, once a structure has been declared, you may declare variables of its type using only its type name, without preceding it with the keyword **struct.** The reason for this difference is that in C, a structure's name does not define a complete type name. In fact, Standard C refers to a structure's name as a *tag*. In C, you must precede the tag with the keyword **struct** when declaring variables. However, in C++, a structure's name is a complete type name and may be used by itself to define variables. Keep in mind, however, that it is still perfectly legal to use the C-style declaration in a C++ program. Since the programs in Part One of this book are valid for both C and C++, they will use the C declaration method. Just remember that C++ allows the shorter form.

When a structure variable (such as **addr_info**) is declared, the compiler automatically allocates sufficient memory to accommodate all of its members. Figure 7-1 shows how **addr_info** appears in memory assuming 1-byte characters and 4-byte long integers.

You may also declare one or more structure variables when you declare a structure. For example,

```
struct addr {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info, binfo, cinfo;
```

defines a structure type called **addr** and declares variables **addr_info**, **binfo**, and **cinfo** of that type. It is important to understand that each structure object contains its own

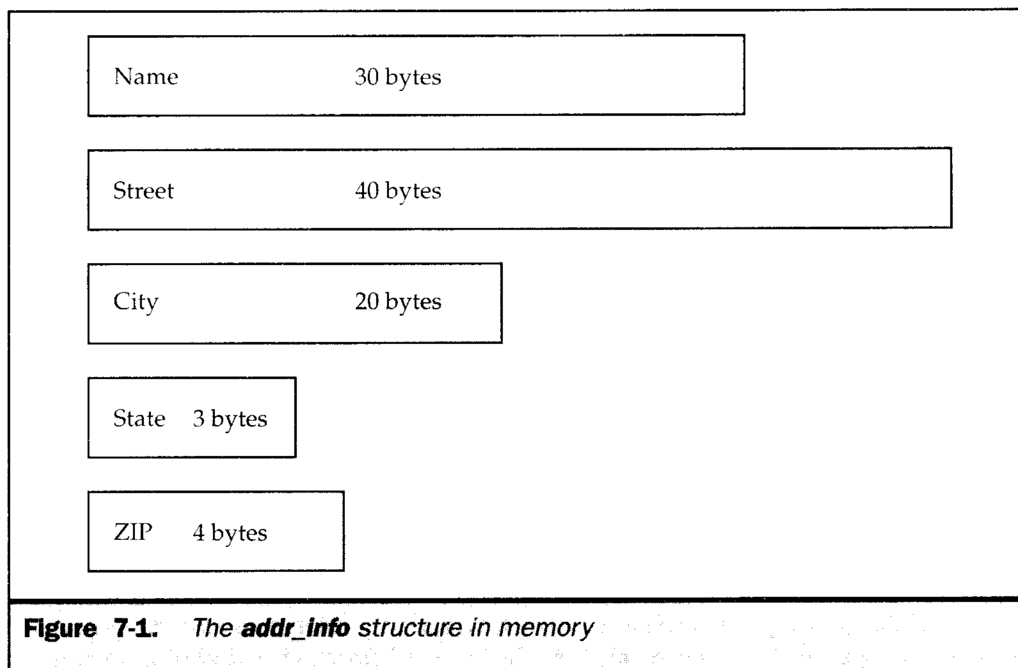

Figure 7-1. *The **addr_info** structure in memory*

copies of the structure's members. For example, the **zip** field of **binfo** is separate from the **zip** field of **cinfo**. Thus, changes to **zip** in **binfo** do not affect the **zip** in **cinfo**.

If you only need one structure variable, the structure type name is not needed. That means that

```
struct {
  char name[30];
  char street[40];
  char city[20];
  char state[3];
  unsigned long int zip;
} addr_info;
```

declares one variable named **addr_info** as defined by the structure preceding it.

The general form of a structure declaration is

struct *struct-type-name* {
 type member-name;
 type member-name;
 type member-name;

} *structure-variables;*

where either *struct-type-name* or *structure-variables* may be omitted, but not both.

Accessing Structure Members

Individual members of a structure are accessed through the use of the . operator (usually called the *dot operator*). For example, the following code assigns the ZIP code 12345 to the **zip** field of the structure variable **addr_info** declared earlier:

```
addr_info.zip = 12345;
```

The structure variable name followed by a period and the member name references that individual member. The general form for accessing a member of a structure is

structure-name.member-name

Therefore, to print the ZIP code on the screen, write

```
printf("%lu", addr_info.zip);
```

This prints the ZIP code contained in the **zip** member of the structure variable **addr_info**.

In the same fashion, the character array **addr_info.name** can be used to call **gets( )**, as shown here:

```
gets(addr_info.name);
```

This passes a character pointer to the start of **name**.

Since **name** is a character array, you can access the individual characters of **addr_info.name** by indexing **name**. For example, you can print the contents of **addr_info.name** one character at a time by using the following code:

```
register int t;

for(t=0; addr_info.name[t]; ++t)
  putchar(addr_info.name[t]);
```

Structure Assignments

The information contained in one structure may be assigned to another structure of the same type using a single assignment statement. That is, you do not need to assign the

value of each member separately. The following program illustrates structure assignments:

```
#include <stdio.h>

int main(void)
{
  struct {
    int a;
    int b;
  } x, y;

  x.a = 10;

  y = x;  /* assign one structure to another */

  printf("%d", y.a);

  return 0;
}
```

After the assignment, **y.a** will contain the value 10.

Arrays of Structures

Perhaps the most common usage of structures is in arrays of structures. To declare an array of structures, you must first define a structure and then declare an array variable of that type. For example, to declare a 100-element array of structures of type **addr**, defined earlier, write

```
struct addr addr_info[100];
```

This creates 100 sets of variables that are organized as defined in the structure **addr**.

To access a specific structure, index the structure name. For example, to print the ZIP code of structure 3, write

```
printf("%lu", addr_info[2].zip);
```

Like all array variables, arrays of structures begin indexing at 0.

Passing Structures to Functions

This section discusses passing structures and their members to functions.

Passing Structure Members to Functions

When you pass a member of a structure to a function, you are actually passing the value of that member to the function. Therefore, you are passing a simple variable (unless, of course, that element is compound, such as an array). For example, consider this structure:

```
struct fred
{
  char x;
  int y;
  float z;
  char s[10];
} mike;
```

Here are examples of each member being passed to a function:

```
func(mike.x);     /* passes character value of x */
func2(mike.y);    /* passes integer value of y */
func3(mike.z);    /* passes float value of z */
func4(mike.s);    /* passes address of string s */
func(mike.s[2]); /* passes character value of s[2] */
```

If you wish to pass the *address* of an individual structure member, put the **&** operator before the structure name. For example, to pass the address of the members of the structure **mike**, write

```
func(&mike.x);     /* passes address of character x */
func2(&mike.y);    /* passes address of integer y */
func3(&mike.z);    /* passes address of float z */
func4(mike.s);     /* passes address of string s */
func(&mike.s[2]); /* passes address of character s[2] */
```

Note that the **&** operator precedes the structure name, not the individual member name. Note also that **s** already signifies an address, so no **&** is required.

Passing Entire Structures to Functions

When a structure is used as an argument to a function, the entire structure is passed using the standard call-by-value method. Of course, this means that any changes made to the contents of the structure inside the function to which it is passed do not affect the structure used as an argument.

When using a structure as a parameter, remember that the type of the argument must match the type of the parameter. For example, in the following program both the argument **arg** and the parameter **parm** are declared as the same type of structure.

```
#include <stdio.h>

/* Define a structure type. */
struct struct_type {
  int a, b;
  char ch;
} ;

void f1(struct struct_type parm);

int main(void)
{
  struct struct_type arg;

  arg.a = 1000;

  f1(arg);

  return 0;
}

void f1(struct struct_type parm)
{
  printf("%d", parm.a);
}
```

As this program illustrates, if you will be declaring parameters that are structures, you must make the declaration of the structure type global so that all parts of your program can use it. For example, had **struct_type** been declared inside **main( )** (for example), then it would not have been visible to **f1( )**.

As just stated, when passing structures, the type of the argument must match the type of the parameter. It is not sufficient for them to simply be physically similar; their type names must match. For example, the following version of the preceding

program is incorrect and will not compile because the type name of the argument used to call **f1( )** differs from the type name of its parameter.

```
/* This program is incorrect and will not compile. */
#include <stdio.h>

/* Define a structure type. */
struct struct_type {
  int a, b;
  char ch;
} ;

/* Define a structure similar to struct_type,
   but with a different name. */
struct struct_type2 {
  int a, b;
  char ch;
} ;

void f1(struct struct_type2 parm);

int main(void)
{
  struct struct_type arg;

  arg.a = 1000;

  f1(arg); /* type mismatch */

  return 0;
}

void f1(struct struct_type2 parm)
{
  printf("%d", parm.a);
}
```

Structure Pointers

C/C++ allows pointers to structures just as it allows pointers to any other type of variable. However, there are some special aspects to structure pointers that you should know.

Declaring a Structure Pointer

Like other pointers, structure pointers are declared by placing * in front of a structure variable's name. For example, assuming the previously defined structure **addr**, the following declares **addr_pointer** as a pointer to data of that type:

```
struct addr *addr_pointer;
```

Remember, in C++ it is not necessary to precede this declaration with the keyword **struct**.

Using Structure Pointers

There are two primary uses for structure pointers: to pass a structure to a function using call by reference, and to create linked lists and other dynamic data structures that rely on dynamic allocation. This chapter covers the first use.

There is one major drawback to passing all but the simplest structures to functions: the overhead needed to push the structure onto the stack when the function call is executed. (Recall that arguments are passed to functions on the stack.) For simple structures with few members, this overhead is not too great. If the structure contains many members, however, or if some of its members are arrays, run-time performance may degrade to unacceptable levels. The solution to this problem is to pass only a pointer to the structure.

When a pointer to a structure is passed to a function, only the address of the structure is pushed on the stack. This makes for very fast function calls. A second advantage, in some cases, is when a function needs to reference the actual structure used as the argument, instead of a copy. By passing a pointer, the function can modify the contents of the structure used in the call.

To find the address of a structure, place the **&** operator before the structure's name. For example, given the following fragment:

```
struct bal {
  float balance;
  char name[80];
} person;

struct bal *p;  /* declare a structure pointer */
```

then

```
p = &person;
```

places the address of the structure **person** into the pointer **p**.

To access the members of a structure using a pointer to that structure, you must use the -> operator. For example, this references the **balance** field:

```
p->balance
```

The -> is usually called the *arrow operator*, and consists of the minus sign followed by a greater-than sign. The arrow is used in place of the dot operator when you are accessing a structure member through a pointer to the structure.

To see how a structure pointer can be used, examine this simple program, which prints the hours, minutes, and seconds on your screen using a software timer.

```
/* Display a software timer. */
#include <stdio.h>

#define DELAY 128000

struct my_time {
  int hours;
  int minutes;
  int seconds;
} ;

void display(struct my_time *t);
void update(struct my_time *t);
void delay(void);

int main(void)
{
  struct my_time systime;

  systime.hours = 0;
  systime.minutes = 0;
  systime.seconds = 0;

  for(;;) {
    update(&systime);
    display(&systime);
  }

  return 0;
}
```

```
void update(struct my_time *t)
{
  t->seconds++;
  if(t->seconds==60) {
    t->seconds = 0;
    t->minutes++;
  }

  if(t->minutes==60) {
    t->minutes = 0;
    t->hours++;
  }

  if(t->hours==24) t->hours = 0;
  delay();
}

void display(struct my_time *t)
{
  printf("%02d:", t->hours);
  printf("%02d:", t->minutes);
  printf("%02d\n", t->seconds);
}

void delay(void)
{
  long int t;

  /* change this as needed */
  for(t=1; t<DELAY; ++t) ;
}
```

The timing of this program is adjusted by changing the definition of **DELAY**.

As you can see, a global structure called **my_time** is defined but no variable is declared. Inside **main( )**, the structure **systime** is declared and initialized to 00:00:00. This means that **systime** is known directly only to the **main( )** function.

The functions **update( )** (which changes the time) and **display( )** (which prints the time) are passed the address of **systime**. In both functions, their arguments are declared as a pointer to a **my_time** structure.

Inside **update( )** and **display( )**, each member of **systime** is accessed via a pointer. Because **update( )** receives a pointer to the **systime** structure, it can update its value.

For example, to set the hours back to 0 when 24:00:00 is reached, **update( )** contains this line of code:

```
if(t->hours==24) t->hours = 0;
```

This tells the compiler to take the address in **t** (which points to **systime** in **main( )**) and use it to reset **hours** to zero.

Remember, use the dot operator to access structure elements when operating on the structure itself. When you have a pointer to a structure, use the arrow operator.

Arrays and Structures Within Structures

A member of a structure may be either a simple or aggregate type. A simple member is one that is of any of the built-in data types, such as integer or character. You have already seen one type of aggregate element: the character arrays used in **addr**. Other aggregate data types include one-dimensional and multidimensional arrays of the other data types, and structures.

A member of a structure that is an array is treated as you might expect from the earlier examples. For example, consider this structure:

```
struct x {
  int a[10][10]; /* 10 x 10 array of ints */
  float b;
} y;
```

To reference integer 3,7 in **a** of structure **y**, write

```
y.a[3][7]
```

When a structure is a member of another structure, it is called a *nested structure*. For example, the structure **address** is nested inside **emp** in this example:

```
struct emp {
  struct addr address; /* nested structure */
  float wage;
} worker;
```

Here, structure **emp** has been defined as having two members. The first is a structure of type **addr**, which contains an employee's address. The other is **wage**, which holds

the employee's wage. The following code fragment assigns 93456 to the **zip** element of **address**.

```
worker.address.zip = 93456;
```

As you can see, the members of each structure are referenced from outermost to innermost. C guarantees that structures can be nested to at least 15 levels. Standard C++ suggests that at least 256 levels of nesting be allowed.

Bit-Fields

Unlike some other computer languages, C/C++ has a built-in feature called a *bit-field* that allows you to access a single bit. Bit-fields can be useful for a number of reasons, such as:

- If storage is limited, you can store several Boolean (true/false) variables in one byte.
- Certain devices transmit status information encoded into one or more bits within a byte.
- Certain encryption routines need to access the bits within a byte.

Although these tasks can be performed using the bitwise operators, a bit-field can add more clarity (and possibly efficiency) to your code.

To access individual bits, C/C++ uses a method based on the structure. In fact, a bit-field is really just a special type of structure member that defines how long, in bits, the field is to be. The general form of a bit-field definition is

struct *struct-type-name* {
  type *name1* : *length;*
  type name2 : *length;*
  .
  .
  .
  type nameN : *length;*
} *variable_list;*

Here, *type* is the type of the bit-field and *length* is the number of bits in the field. A bit-field must be declared as an integral or enumeration type. Bit-fields of length 1 should be declared as **unsigned**, because a single bit cannot have a sign.

Bit-fields are frequently used when analyzing input from a hardware device. For example, the status port of a serial communications adapter might return a status byte organized like this:

Bit	Meaning When Set
0	Change in clear-to-send line
1	Change in data-set-ready
2	Trailing edge detected
3	Change in receive line
4	Clear-to-send
5	Data-set-ready
6	Telephone ringing
7	Received signal

You can represent the information in a status byte using the following bit-field:

```
struct status_type {
  unsigned delta_cts: 1;
  unsigned delta_dsr: 1;
  unsigned tr_edge:   1;
  unsigned delta_rec: 1;
  unsigned cts:       1;
  unsigned dsr:       1;
  unsigned ring:      1;
  unsigned rec_line:  1;
} status;
```

You might use a routine similar to that shown here to enable a program to determine when it can send or receive data.

```
status = get_port_status();
if(status.cts) printf("clear to send");
if(status.dsr) printf("data ready");
```

To assign a value to a bit-field, simply use the form you would use for any other type of structure element. For example, this code fragment clears the **ring** field:

```
status.ring = 0;
```

As you can see from this example, each bit-field is accessed with the dot operator. However, if the structure is referenced through a pointer, you must use the –> operator.

You do not have to name each bit-field. This makes it easy to reach the bit you want, bypassing unused ones. For example, if you only care about the **cts** and **dsr** bits, you could declare the **status_type** structure like this:

```
struct status_type {
  unsigned :    4;
  unsigned cts: 1;
  unsigned dsr: 1;
} status;
```

Also, notice that the bits after **dsr** do not need to be specified if they are not used.

It is valid to mix normal structure members with bit-fields. For example,

```
struct emp {
  struct addr address;
  float pay;
  unsigned lay_off:    1; /* lay off or active */
  unsigned hourly:     1; /* hourly pay or wage */
  unsigned deductions: 3; /* IRS deductions */
};
```

defines an employee record that uses only 1 byte to hold three pieces of information: the employee's status, whether the employee is salaried, and the number of deductions. Without the bit-field, this information would have taken 3 bytes.

Bit-fields have certain restrictions. You cannot take the address of a bit-field. Bit-fields cannot be arrayed. They cannot be declared as **static**. You cannot know, from machine to machine, whether the fields will run from right to left or from left to right; this implies that any code using bit-fields may have some machine dependencies. Other restrictions may be imposed by various specific implementations.

Unions

A *union* is a memory location that is shared by two or more different types of variables. A union provides a way of interpreting the same bit pattern in two or more different ways. Declaring a **union** is similar to declaring a structure. Its general form is

union *union-type-name* {
  type member-name;

type member-name;
type member-name;
.
.
.
} *union-variables;*

For example:

```
union u_type {
  int i;
  char ch;
};
```

This declaration does not create any variables. You may declare a variable either by placing its name at the end of the declaration or by using a separate declaration statement. In C, to declare a **union** variable called **cnvt** of type **u_type** using the definition just given, write

```
union u_type cnvt;
```

When declaring **union** variables in C++, you need use only the type name— you don't need to precede it with the keyword **union**. For example, this is how **cnvt** is declared in C++:

```
u_type cnvt;
```

In C++, preceding this declaration with the keyword **union** is allowed, but redundant. In C++, the name of a **union** defines a complete type name. In C, a union name is its tag and it must be preceded by the keyword **union**. (This is similar to the situation with structures described earlier.) However, since the programs in this chapter are valid for both C and C++, the C-style declaration form will be used.

In **cnvt**, both integer **i** and character **ch** share the same memory location. Of course, **i** occupies 2 bytes (assuming 2-byte integers) and **ch** uses only 1. Figure 7-2 shows how **i** and **ch** share the same address. At any point in your program, you can refer to the data stored in a **cnvt** as either an integer or a character.

When a **union** variable is declared, the compiler automatically allocates enough storage to hold the largest member of the **union**. For example (assuming 2-byte integers), **cnvt** is 2 bytes long so that it can hold **i**, even though **ch** requires only 1 byte.

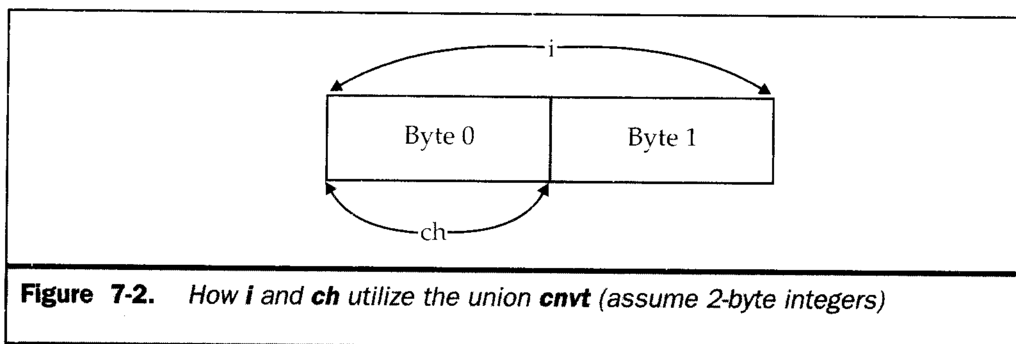

Figure 7-2. *How **i** and **ch** utilize the union **cnvt** (assume 2-byte integers)*

To access a member of a **union**, use the same syntax that you would use for structures: the dot and arrow operators. If you are operating on the **union** directly, use the dot operator. If the **union** is accessed through a pointer, use the arrow operator. For example, to assign the integer 10 to element **i** of **cnvt**, write

```
cnvt.i = 10;
```

In the next example, a pointer to **cnvt** is passed to a function:

```
void func1(union u_type *un)
{
  un->i = 10; /* assign 10 to cnvt through
                 a pointer */
}
```

Unions are used frequently when specialized type conversions are needed because you can refer to the data held in the **union** in fundamentally different ways. For example, you may use a **union** to manipulate the bytes that comprise a **double** in order to alter its precision or to perform some unusual type of rounding.

To get an idea of the usefulness of a **union** when nonstandard type conversions are needed, consider the problem of writing a short integer to a disk file. The C/C++ standard library defines no function specifically designed to write a short integer to a file. While you can write any type of data to a file using **fwrite( )**, using **fwrite( )** incurs excessive overhead for such a simple operation. However, using a **union** you can easily create a function called **putw( )**, which writes the binary representation of a short integer to a file one byte at a time. (This example assumes that short integers are 2 bytes long.) To see how, first create a **union** consisting of one short integer and a 2-byte character array:

```
union pw {
  short int i;
```

```
  char ch[2];
};
```

Now, you can use **pw** to create the version of **putw( )** shown in the following program.

```
#include <stdio.h>

union pw {
  short int i;
  char ch[2];
};

int putw(short int num, FILE *fp);

int main(void)
{
  FILE *fp;

  fp = fopen("test.tmp", "wb+");

  putw(1000, fp);  /* write the value 1000 as an integer */
  fclose(fp);

  return 0;
}

int putw(short int num, FILE *fp)
{
  union pw word;

  word.i = num;

  putc(word.ch[0], fp); /* write first half */
  return putc(word.ch[1], fp); /* write second half */
}
```

Although **putw( )** is called with a short integer, it can still use the standard function **putc( )** to write each byte in the integer to a disk file one byte at a time.

Note *C++ supports a special type of union called an anonymous union which is discussed in Part Two of this book.*

Enumerations

An *enumeration* is a set of named integer constants that specify all the legal values a variable of that type may have. Enumerations are common in everyday life. For example, an enumeration of the coins used in the United States is

penny, nickel, dime, quarter, half-dollar, dollar

Enumerations are defined much like structures; the keyword **enum** signals the start of an enumeration type. The general form for enumerations is

enum *enum-type-name* { *enumeration list* } *variable_list;*

Here, both the type name and the variable list are optional. (But at least one must be present.) The following code fragment defines an enumeration called **coin**:

```
enum coin { penny, nickel, dime, quarter,
            half_dollar, dollar};
```

The enumeration type name can be used to declare variables of its type. In C, the following declares **money** to be a variable of type **coin**.

```
enum coin money;
```

In C++, the variable **money** may be declared using this shorter form:

```
coin money;
```

In C++, an enumeration name specifies a complete type. In C, an enumeration name is its tag and it requires the keyword **enum** to complete it. (This is similar to the situation as it applies to structures and unions, described earlier.)

Given these declarations, the following types of statements are perfectly valid:

```
money = dime;
if(money==quarter) printf("Money is a quarter.\n");
```

The key point to understand about an enumeration is that each of the symbols stands for an integer value. As such, they may be used anywhere that an integer may be used. Each symbol is given a value one greater than the symbol that precedes it. The value of the first enumeration symbol is 0. Therefore,

```
printf("%d %d", penny, dime);
```

displays **0 2** on the screen.

You can specify the value of one or more of the symbols by using an initializer. Do this by following the symbol with an equal sign and an integer value. Symbols that appear after initializers are assigned values greater than the previous initialization value. For example, the following code assigns the value of 100 to **quarter**:

```
enum coin { penny, nickel, dime, quarter=100,
            half_dollar, dollar};
```

Now, the values of these symbols are

penny	0
nickel	1
dime	2
quarter	100
half_dollar	101
dollar	102

One common but erroneous assumption about enumerations is that the symbols can be input and output directly. This is not the case. For example, the following code fragment will not perform as desired:

```
/* this will not work */
money = dollar;
printf("%s", money);
```

Remember, **dollar** is simply a name for an integer; it is not a string. For the same reason, you cannot use this code to achieve the desired results:

```
/* this code is wrong */
strcpy(money, "dime");
```

That is, a string that contains the name of a symbol is not automatically converted to that symbol.

Actually, creating code to input and output enumeration symbols is quite tedious (unless you are willing to settle for their integer values). For example, you need the following code to display, in words, the kind of coins that **money** contains:

```
switch(money) {
  case penny: printf("penny");
    break;
  case nickel: printf("nickel");
    break;
  case dime: printf("dime");
    break;
  case quarter: printf("quarter");
    break;
  case half_dollar: printf("half_dollar");
    break;
  case dollar: printf("dollar");
}
```

Sometimes you can declare an array of strings and use the enumeration value as an index to translate that value into its corresponding string. For example, this code also outputs the proper string:

```
char name[][12]={
  "penny",
  "nickel",
  "dime",
  "quarter",
  "half_dollar",
  "dollar"
};
printf("%s", name[money]);
```

Of course, this only works if no symbol is initialized, because the string array must be indexed starting at 0 in strictly ascending order using increments of 1.

Since enumeration values must be converted manually to their human-readable string values for I/O operations, they are most useful in routines that do not make such conversions. An enumeration is often used to define a compiler's symbol table, for example. Enumerations are also used to help prove the validity of a program by providing a compile-time redundancy check confirming that a variable is assigned only valid values.

Using sizeof to Ensure Portability

You have seen that structures and unions can be used to create variables of different sizes, and that the actual size of these variables may change from machine to machine. The **sizeof** operator computes the size of any variable or type and can help eliminate machine-dependent code from your programs. This operator is especially useful where structures or unions are concerned.

For the following discussion, assume an implementation, common to many C/C++ compilers, that has the sizes for data types shown here:

Type	Size in Bytes
char	1
int	4
double	8

Therefore, the following code will print the numbers 1, 4, and 8 on the screen:

```
char ch;
int i;
double f;

printf("%d", sizeof(ch));
printf("%d", sizeof(i));
printf("%d", sizeof(f));
```

The size of a structure is equal to *or greater than* the sum of the sizes of its members. For example,

```
struct s {
  char ch;
  int i;
  double f;
} s_var;
```

Here, **sizeof(s_var)** is at least 13 (8 + 4 + 1). However, the size of **s_var** might be greater because the compiler is allowed to pad a structure in order to achieve word or paragraph alignment. (A paragraph is 16 bytes.) Since the size of a structure may be greater than the sum of the sizes of its members, you should always use **sizeof** when you need to know the size of a structure.

Since **sizeof** is a compile-time operator, all the information necessary to compute the size of any variable is known at compile time. This is especially meaningful for **unions**, because the size of a **union** is always equal to the size of its largest member. For example, consider

```
union u {
  char ch;
  int i;
  double f;
} u_var;
```

Here, the **sizeof(u_var)** is 8. At run time, it does not matter what **u_var** is actually holding. All that matters is the size of its largest member, because any **union** must be as large as its largest element.

typedef

You can define new data type names by using the keyword **typedef**. You are not actually *creating* a new data type, but rather defining a new name for an existing type. This process can help make machine-dependent programs more portable. If you define your own type name for each machine-dependent data type used by your program, then only the **typedef** statements have to be changed when compiling for a new environment. **typedef** also can aid in self-documenting your code by allowing descriptive names for the standard data types. The general form of the **typedef** statement is

typedef *type newname;*

where *type* is any valid data type and *newname* is the new name for this type. The new name you define is in addition to, not a replacement for, the existing type name.

For example, you could create a new name for **float** by using

```
typedef float balance;
```

This statement tells the compiler to recognize **balance** as another name for **float**. Next, you could create a **float** variable using **balance**:

```
balance over_due;
```

Here, **over_due** is a floating-point variable of type **balance**, which is another word for **float**.

Now that **balance** has been defined, it can be used in another **typedef**. For example,

```
typedef balance overdraft;
```

tells the compiler to recognize **overdraft** as another name for **balance**, which is another name for **float**.

Using **typedef** can make your code easier to read and easier to port to a new machine, but you are not creating a new physical type.