Here is the reason why the compiler complains. Since p is an ordinary pointer to int, we could use it later in an expression such as ++*p to change the stored value of a, violating the concept that a is constant. If, however, we write

```
const int   a = 7;
const int   *p = &a;
```

then the compiler will be happy. The last declaration is read "p is a pointer to a constant int and its initial value is the address of a." Note that p itself is not constant. We can assign to it some other address. We may not, however, assign a value to *p. The object pointed to by p should not be modified.

Suppose we want p itself to be constant, but not a. This is achieved with the following declarations:

```
int         a;
int * const   p = &a;
```

We read the last declaration as "p is a constant pointer to int, and its initial value is the address of a." Thereafter, we may not assign a value to p, but we may assign a value to *p. Now consider

```
const int         a = 7;
const int * const   p = &a;
```

The last declaration tells the compiler that p is a constant pointer to a constant int. Neither p nor *p can be assigned to, incremented, or decremented.

In contrast to const, the type qualifier volatile is seldom used. A volatile object is one that can be modified in some unspecified way by the hardware. Now consider the declaration

```
extern const volatile int   real_time_clock;
```

The extern means "look for it elsewhere, either in this file or in some other file." The qualifier volatile indicates that the object may be acted on by the hardware. Because const is also a qualifier, the object may not be assigned to, incremented, or decremented within the program. The hardware can change the clock, but the code cannot.

Summary

1 The brackets [] are used in a declaration to tell the compiler that an identifier is an array. The integral constant expression in the brackets specifies the size of the array. For example, the declaration

```
int   a[100];
```

causes the compiler to allocate contiguous space in memory for 100 ints. The elements of the array are numbered from 0 to 99. The array name a by itself is a constant pointer; its value is the base address of the array.

2 A pointer variable takes addresses as values. Some typical values are NULL, addresses of variables, string constants, and pointer values, or addresses, returned from functions such as calloc(). If allocation fails—for example, the system free store (heap) is exhausted—then NULL is returned.

3 The address operator & and the indirection or dereferencing operator * are unary operators with the same precedence and right to left associativity as other unary operators. If v is a variable, then the expression

*&v is equivalent to v

4 Pointers are used as formal parameters in headers to function definitions to effect "call-by-reference." When addresses of variables are passed as arguments, they can be dereferenced in the body of the function to change the values of variables in the calling environment.

5 In C, arrays and pointers are closely related topics. If a is an array and i is an int, then the expression

a[i] is equivalent to *(a + i)

These expressions can be used to access elements of the array. The expression a + i is an example of pointer arithmetic. Its value is the address of the element of the array that is i elements beyond a itself. That is, a + i is equivalent to &a[i].

6 In the header to a function definition, the declaration of a parameter as an array is equivalent to its declaration as a pointer. For example,

```
int a[]      is equivalent to      int *a
```

This equivalence does *not* hold elsewhere.

7 When an array is passed as an argument to a function, a pointer is actually passed. The array elements themselves are not copied.

8 Strings are one-dimensional arrays of characters. By convention, they are terminated with the null character \0, which acts as the end-of-string sentinel.

9 The standard library contains many useful string-handling functions. For example, strlen() returns the length of a string, and strcat() concatenates two strings.

10 Arrays of any type can be created, including arrays of arrays. For example,

```
double   a[3][7];
```

declares a to be an array of "array of 7 doubles." The elements of a are accessed by expressions such as a[i][j]. The base address of the array is &a[0][0], not a. The array name a by itself is equivalent to &a[0].

11 In the header to a function definition, the declaration of a multidimensional array must have all sizes specified except the first. This allows the compiler to generate the correct storage mapping function.

12 Arguments to main() are typically called argc and argv. The value of argc is the number of command line arguments. The elements of the array argv are addresses of the command line arguments. We can think of argv as an array of strings.

13 Ragged arrays are constructed from arrays of pointers. The elements of the array can point to arrays with different sizes.

14 Like an array name, a function name that is passed as an argument is treated as a pointer. In the body of the function the pointer can be used to call the function in the normal way, or it can be explicitly dereferenced.

15 The type qualifiers const and volatile have been added to ANSI C. They are not available in traditional C.