Now suppose that p is made to point to a dynamically allocated block of memory:

```
p = malloc(sizeof(int));
```

(A similar situation would arise if p were assigned the address of a variable or an array element.) Normally it would be legal to copy p into q and then modify the integer through q:

```
q = p;
*q = 0;    /* causes undefined behavior */
```

Because p is a restricted pointer, however, the effect of executing the statement *q = 0; is undefined. By making p and q point to the same object, we caused *p and *q to be aliases.

extern storage class ➤18.2

blocks ➤10.3

If a restricted pointer p is declared as a local variable without the extern storage class, restrict applies only to p when the block in which p is declared is being executed. (Note that the body of a function is a block.) restrict can be used with function parameters of pointer type, in which case it applies only when the function is executing. When restrict is applied to a pointer variable with file scope, however, the restriction lasts for the entire execution of the program.

file scope ➤10.2

The exact rules for using restrict are rather complex; see the C99 standard for details. There are even situations in which an alias created from a restricted pointer is legal. For example, a restricted pointer p can be legally copied into another restricted pointer variable q, provided that p is local to a function and q is defined inside a block nested within the function's body.

<string.h> header ➤23.6

To illustrate the use of restrict, let's look at the memcpy and memmove functions, which belong to the <string.h> header. memcpy has the following prototype in C99:

```
void *memcpy(void * restrict s1, const void * restrict s2,
             size_t n);
```

memcpy is similar to strcpy, except that it copies bytes from one object to another (strcpy copies characters from one string into another). s2 points to the data to be copied, s1 points to the destination of the copy, and n is the number of bytes to be copied. The use of restrict with both s1 and s2 indicates that the source of the copy and the destination shouldn't overlap. (It doesn't *guarantee* that they don't overlap, however.)

In contrast, restrict doesn't appear in the prototype for memmove:

```
void *memmove(void *s1, const void *s2, size_t n);
```

memmove does the same thing as memcpy: it copies bytes from one place to another. The difference is that memmove is guaranteed to work even if the source and destination overlap. For example, we could use memmove to shift the elements of an array by one position:

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