```
struct point create_point(int x, int y)
{
  return (struct point) {x, y};
}
```

This function works correctly, because the object created by the compound literal will be copied when the function returns. The original object will no longer exist, but the copy will remain. Now suppose that we change the function slightly:

```
struct point *create_point(int x, int y)
{
  return &(struct point) {x, y};
}
```

This version of `create_point` suffers from undefined behavior, because it returns a pointer to an object that has automatic storage duration and won't exist after the function returns.

Now let's return to the question we started with: Why are selection statements and iteration statements considered to be blocks? Consider the following example:

```
/* Example 1 - if statement without braces */

double *coefficients, value;

if (polynomial_selected == 1)
  coefficients = (double[3]) {1.5, -3.0, 6.0};
else
  coefficients = (double[3]) {4.5, 1.0, -3.5};
value = evaluate_polynomial(coefficients);
```

This program fragment apparently behaves in the desired fashion (but read on). `coefficients` will point to one of two objects created by compound literals. and this object will still exist at the time `evaluate_polynomial` is called. Now consider what happens if we put braces around the "inner" statements—the ones controlled by the `if` statement:

```
/* Example 2 - if statement with braces */

double *coefficients, value;

if (polynomial_selected == 1) {
  coefficients = (double[3]) {1.5, -3.0, 6.0};
} else {
  coefficients = (double[3]) {4.5, 1.0, -3.5};
}
value = evaluate_polynomial(coefficients);
```

Now we're in trouble. Each compound literal causes an object to be created, but that object exists only within the block formed by the braces that enclose the statement in which the literal appears. By the time `evaluate_polynomial` is called, `coefficients` points to an object that no longer exists. The result: undefined behavior.