```
  return 0;
}
```

Notice that `main` contains a variable named `n` even though `is_prime`'s parameter is also named `n`. In general, a function may declare a variable with the same name as a variable in another function. The two variables represent different locations in memory, so assigning a new value to one variable doesn't change the other. (This property extends to parameters as well.) Section 10.1 discusses this point in more detail.

As `is_prime` demonstrates, a function may have more than one `return` statement. However, we can execute just one of these statements during a given call of the function, because reaching a `return` statement causes the function to return to where it was called. We'll learn more about the `return` statement in Section 9.4.

9.2 Function Declarations

In the programs in Section 9.1, the definition of each function was always placed *above* the point at which it was called. In fact, C doesn't require that the definition of a function precede its calls. Suppose that we rearrange the `average.c` program by putting the definition of `average` *after* the definition of `main`:

```
#include <stdio.h>

int main(void)
{
  double x, y, z;

  printf("Enter three numbers: ");
  scanf("%lf%lf%lf", &x, &y, &z);
  printf("Average of %g and %g: %g\n", x, y, average(x, y));
  printf("Average of %g and %g: %g\n", y, z, average(y, z));
  printf("Average of %g and %g: %g\n", x, z, average(x, z));

  return 0;
}

double average(double a, double b)
{
  return (a + b) / 2;
}
```

When the compiler encounters the first call of `average` in `main`, it has no information about `average`: it doesn't know how many parameters `average` has, what the types of these parameters are, or what kind of value `average` returns. Instead of producing an error message, though, the compiler assumes that `average` returns an `int` value (recall from Section 9.1 that the return type of a