```
/* Read a number and determine if it is prime.
*/
#include <stdio.h>

int isprime(int n);

int
main(int argc, char *argv[]) {
    int n;
    printf("Enter a number n: ");
    scanf("%d", &n);
    if (isprime(n)) {
        printf("%d is a prime number\n", n);
    } else {
        printf("%d is not a prime number\n", n);
    }
    return 0;
}

/* Determine whether n is prime. */
int
isprime(int n) {
    int divisor;
    if (n<2) {
        return 0;
    }
    for (divisor=2; divisor*divisor<=n; divisor++) {
           if (n%divisor==0) {
               /* factor found, so can't be prime */
               return 0;
           }
    }
    /* no factors, so must be prime */
    return 1;
}
```

**Figure 5.4:** A function `isprime`, and a `main` function that calls it. This program is complete within a single file.

`break` statement that was used in the earlier version of this computation (Figure 4.10 on page 55), terminating both the loop and the function. The second observation is that the returned value is used in place of the explicit `isprime` flag of the previous version. The third observation is that the calling program in Figure 5.4 is unable to access the value of the local variable `divisor` the way it did in Figure 4.10, and it is not possible for the program to print out a pair of factors of `n`. A function can only return a single value; and all local variables and argument variables are discarded at the time the function returns. If the function is then called a second time, the argument variables are re-initialized to the new argument expressions, and the local variables are recreated in an uninitialized state. They do *not* resume their final values from the previous call to that function.

```
mac: ls
func.c    prog.c
mac: gcc -Wall -ansi -c func.c
mac: gcc -Wall -ansi -c prog.c
mac: ls
func.c    func.o    prog.c    prog.o
mac: gcc -Wall -o prog prog.o func.o
mac: ls
func.c    func.o    prog      prog.c    prog.o
mac: ./prog
<<< whatever the program does >>>
mac:
```

**Figure 5.5:** Example showing separate compilation. After the three `gcc` commands, the file `prog` contains a self-contained and executable program. Neither `func.o` nor `prog.o` can be executed.

The simplest way to manage a program that uses functions is to put all components of the program into a single file and then compile that file.

When a program is more complex, or when several programs are sharing a common pool of useful functions, inserting the text of the function definition in each file that needs it is wasteful. To avoid the redundancy, *separate compilation* is used. Figure 5.5 shows how this works, assuming (as in Section 1.3) that the `gcc` compiler is available. Two source files are manipulated in the interaction shown in the figure: `func.c`, which is assumed to contain one or more function definitions; and file `prog.c`, which is assumed to contain a program that uses those functions. For a program to be executable once it is compiled, whether it is all in a single file or spread across multiple files, exactly one `main` function should be present.

In the example, each of the two C files is first compiled separately. Use of the `-c` flag to the C compiler instructs it to simply compile the code in the file to make an *object file* (with a "`.o`" extension), without attempting to locate functions that are used but not defined. The third `gcc` instruction then combines the two "`.o`" files to build an executable program called `prog`. The complete compilation could also have been achieved using a single command-line:

```
gcc -Wall -ansi -o prog prog.c func.c
```

Separate compilation allows efficient reuse of software modules. Each source file contains a group of logically related functions, and is compiled independently of other modules. A final linking step then creates the executable program.

If separate compilation is used, keeping track of which files have been edited can become a chore. And in a big program, split over a large number of separate source files, it is faster to only recompile files that have been edited since the last compilation, and then rebuild the final executable. Fortunately, there is a tool to help with this, called `make`, that examines the "last modified" time-stamps on a set of files