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

int isprime(int);

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 argument 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 5.3: A function `isprime`, and a `main` function that calls it. This program is complete within a single file, and can be compiled and executed.

variables are recreated in an uninitialized state. They do *not* resume their final values from the previous call to that function.

> 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 5.4 shows how this works, assuming (as in Section 1.3) that the `gcc` compiler is available. Two files are manipulated in the interaction shown in the figure: `func.c`, which is assumed to contain a function definition (perhaps function `savings_plan`,

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

Figure 5.4: 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.

as shown in Figure 5.1 on page 64); and file `prog.c`, which is assumed to contain a program that uses that function (for example, as shown in Figure 5.2 on page 66).

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
```

However, in a big program, comprising a large number of separate source files, it is more economical to only recompile the files that have been edited since the last compilation, and then rebuild the final executable.

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.

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.

If separate compilation is used, keeping track of which files have been edited can become a chore. Fortunately, there is a tool for helping with this, called `make`, that examines the "last modified" time-stamps on a set of files and determines which components need recompilation. It is not appropriate to provide a detailed discussion of `make` here, but when you start working with bigger programs, you are going to need to know about it and how to structure the controlling `makefile`.