in the program but called as `read_cahr`, the linker will report that `read_cahr` is missing.

- *Missing files.* If the linker can't find the functions that are in file `foo.c`, it may not know about the file. Check the makefile or project file to make sure that `foo.c` is listed there.
- *Missing libraries.* The linker may not be able to find all library functions used in the program. A classic example occurs in UNIX programs that use the `<math.h>` header. Simply including the header in a program may not be enough: many versions of UNIX require that the `-lm` option be specified when the program is linked, causing the linker to search a system file that contains compiled versions of the `<math.h>` functions. Failing to use this option may cause "undefined reference" messages during linking.

Rebuilding a Program

During the development of a program, it's rare that we'll need to compile all its files. Most of the time, we'll test the program, make a change, then build the program again. To save time, the rebuilding process should recompile only those files that might be affected by the latest change.

Let's assume that we've designed our program in the way outlined in Section 15.3, with a header file for each source file. To see how many files will need to be recompiled after a change, we need to consider two possibilities.

The first possibility is that the change affects a single source file. In that case, only that file must be recompiled. (After that, the entire program will need to be relinked, of course.) Consider the `justify` program. Suppose that we decide to condense the `read_char` function in `word.c` (changes are marked in **bold**):

```
int read_char(void)
{
  int ch = getchar();

  return (ch == '\n' || ch == '\t') ? ' ' : ch;
}
```

This modification doesn't affect `word.h`, so we need only recompile `word.c` and relink the program.

The second possibility is that the change affects a header file. In that case, we should recompile all files that include the header file, since they could potentially be affected by the change. (Some of them might not be, but it pays to be conservative.)

As an example, consider the `read_word` function in the `justify` program. Notice that `main` calls `strlen` immediately after calling `read_word`, in order to determine the length of the word that was just read. Since `read_word` already knows the length of the word (`read_word`'s `pos` variable keeps track of the length), it seems silly to use `strlen`. Modifying `read_word` to return the word's length is easy. First, we change the prototype of `read_word` in `word.h`: