Chapter 55: Inlining

Section 55.1: Inlining functions used in more than one source file

For small functions that get called often, the overhead associated with the function call can be a significant fraction of the total execution time of that function. One way of improving performance, then, is to eliminate the overhead.

In this example we use four functions (plus `main()`) in three source files. Two of those (`plusfive()` and `timestwo()`) each get called by the other two located in "source1.c" and "source2.c". The `main()` is included so we have a working example.

main.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int main(void) {
    int start = 3;
    int intermediate = complicated1(start);
    printf("First result is %d\n", intermediate);
    intermediate = complicated2(start);
    printf("Second result is %d\n", intermediate);
    return 0;
}
```

source1.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated1(int input) {
    int tmp = timestwo(input);
    tmp = plusfive(tmp);
    return tmp;
}
```

source2.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include "headerfile.h"

int complicated2(int input) {
    int tmp = plusfive(input);
    tmp = timestwo(tmp);
    return tmp;
}
```

headerfile.h:

```c
#ifndef HEADERFILE_H
#define HEADERFILE_H

int complicated1(int input);
int complicated2(int input);

inline int timestwo(int input) {
  return input * 2;
}
inline int plusfive(int input) {
  return input + 5;
}

#endif
```

Functions `timestwo` and `plusfive` get called by both `complicated1` and `complicated2`, which are in different "translation units", or source files. In order to use them in this way, we have to define them in the header.

Compile like this, assuming gcc:

```
cc -O2 -std=c99 -c -o main.o main.c
cc -O2 -std=c99 -c -o source1.o source1.c
cc -O2 -std=c99 -c -o source2.o source2.c
cc main.o source1.o source2.o -o main
```

We use the -O2 optimization option because some compilers don't inline without optimization turned on.

The effect of the **`inline`** keyword is that the function symbol in question is not emitted into the object file. Otherwise an error would occur in the last line, where we are linking the object files to form the final executable. If we would not have **`inline`**, the same symbol would be defined in both `.o` files, and a "multiply defined symbol" error would occur.

In situations where the symbol is actually needed, this has the disadvantage that the symbol is not produced at all. There are two possibilities to deal with that. The first is to add an extra **`extern`** declaration of the inlined functions in exactly one of the `.c` files. So add the following to `source1.c`:

```
extern int timestwo(int input);
extern int plusfive(int input);
```

The other possibility is to define the function with `static` **`inline`** instead of **`inline`**. This method has the drawback that eventually a copy of the function in question may be produced in **every** object file that is produced with this header.