Functions and projects

Recall the temperature conversion program, `conversion.c`.

conversion.c

```c
#include <stdio.h>

int main(int argc, char **argv)
{
        int fahrenheit, celsius;
        celsius = -40;
        fahrenheit = 9 * celsius / 5 + 32;
        printf("%d Celsius is %d Fahrenheit.\n", celsius, fahrenheit);
        return 0;
}
```

The `main` function here demonstrates the syntax for defining functions, and the call to `printf` the syntax for calling them. Let us emphasize both by creating a new function: Factor the temperature conversion math out from its input values and what is done with the result, into its own reusable function, `convert`.

conversion2.c

```c
#include <stdio.h>

int convert(int celsius)
{
        return 9 * celsius / 5 + 32;
}

int main(int argc, char **argv)
{
        int fahrenheit, celsius;
        celsius = -40;
        fahrenheit = convert(celsius);
        printf("%d Celsius is %d Fahrenheit.\n", celsius, fahrenheit);
        return 0;
}
```

The order of these definitions is very important. If `convert` were to appear after `main` in the file, even with everything else unchanged, `gcc` would complain.

```
$ gcc -Wall -g -o conversion2 conversion2.c
conversion2.c: In function main:
conversion2.c:7: warning: implicit declaration of function convert
```

The compiler has to know about a function before it is called, or else it won't know how to make sure it is being used correctly. However, it doesn't need to know the whole definition, just the arguments and return type. A “prototype” is simply a function *declaration* without the associated *definition*, and is enough for the compiler can check and compile the code correctly.

conversion3.c

```c
#include <stdio.h>

int convert(int celsius);

int main(int argc, char **argv)
{
	int fahrenheit, celsius;
	celsius = -40;
	fahrenheit = convert(celsius);
	printf("%d Celsius is %d Fahrenheit.\n", celsius, fahrenheit);
	return 0;
}

int convert(int celsius)
{
	return 9 * celsius / 5 + 32;
}
```

Prototypes for a larger project or library are generally collected together into one or more header files. Whereas actual C code containing definitions generally resides in files with a `.c` extension, C language header files generally reside in files with a `.h` extension. Furthermore, C supports what is known as “separate compilation”—not all of the definitions need to reside in the same file, so long as the compiler knows all of the prototypes it needs to compile each one separately; the resulting individual object files are not executable independently, but the compiler can then link them together into one executable.

One excellent way to organize a project is to put each function in its own `.c` file, and put all of the prototypes together in a `.h` file. The preprocessor `#include` directive, which we have already seen for bringing in system-wide headers, is also used to bring in local headers. Thus, we can break the conversion program into three files and then compile all of them together. The `main` function goes, alone, in `conversion4.c`.

conversion4.c

```c
#include <stdio.h>
#include "convert.h"

int main(int argc, char **argv)
{
	int fahrenheit, celsius;
	celsius = -40;
	fahrenheit = convert(celsius);
	printf("%d Celsius is %d Fahrenheit.\n", celsius, fahrenheit);
	return 0;
}
```

The name of the header file is in quotes, rather than brackets, which tells the preprocessor to look for it in the same directory as the `.c` file, instead of looking for it in the system-wide directories where headers like `stdio.h` are installed.

The header itself, `convert.h`, only contains the prototypes (in this case, just the one prototype).

convert.h

```
int convert(int celsius);
```

Finally, the temperature-conversion function can reside in its own file, `convert.c`. (It is not a coincidence that the file is named after the function. The compiler does not care in the slightest, but you as a human want to remain sane.)

convert.c

```
#include "convert.h"

int convert(int celsius)
{
        return 9 * celsius / 5 + 32;
}
```

Now, the two C files are compiled separately and then linked together before there is an executable. The `-c` flag tells `gcc` to compile but not link; the intermediate files thus produced traditionally bear the `.o` extension.

```
$ gcc -Wall -g -c -o conversion4.o conversion4.c
$ gcc -Wall -g -c -o convert.o convert.c
$ gcc -Wall -g -o conversion4 conversion4.o convert.o
$ ./conversion4
-40 Celsius is -40 Fahrenheit.
```

Notice that `convert.c` includes the header even though it does not actually need it. This is a good idea for two reasons. The first is that a larger project the functions will probably call each other, and thus need to know about each other anyway. The second even applies here, and is simply that it enables the compiler to detect if the prototype and the actual definition do not match. For instance, if we change the argument to a `double`, `gcc` issues an error.

```
$ gcc -Wall -g -c -o convert.o convert.c
convert.c:3: error: conflicting types for convert
convert.h:1: note: previous declaration of convert was here
```

Without including the header file, this error would be silently and very confusingly accepted.

Actually, this demonstration of breaking `convert.c` also demonstrated the main advantage of separate compilation: Changes in one file require it to be recompiled, as always, but there is no need to recompile any of the other, unchanged files, just relink to bring in the one updated object file. Thus, after fixing `convert.c`, all that is required is:

```
$ gcc -Wall -g -c -o convert.o convert.c
$ gcc -Wall -g -o conversion4 conversion4.o convert.o
$ ./conversion4
-40 Celsius is -40 Fahrenheit.
```

There is a UNIX command called `make` that exploits this capability even further. You write a file named `Makefile`, which just sits in the same directory as your project, and describes what source files are used to build which object files, what headers they depend on, and so forth. After you invest the effort to create it, however, you can always build your entire project by simply typing `make`. It will parse the `Makefile`, determine what files have been changed based on their timestamps, and issue only those commands required to bring everything up to date.

In fact, `make` is so optimized for C projects that it has default rules for invoking `gcc` for compiling intermediate objects, linking them, *et cetera*. You can customize its default invocation by providing `gcc` options in the `CFLAGS` variable in the `Makefile`. Thus, here is a `Makefile` that builds `conversion4` as above, automatically.

`Makefile`

```
CFLAGS=-Wall -g

conversion4: conversion4.o convert.o
conversion4.o: conversion4.c convert.h
convert.o: convert.c convert.h

.PHONY: clean
clean:
	rm -f conversion4 conversion4.o convert.o
```

The stuff at the bottom about `clean` adds a special extra thing you can "build". If you run the command `make clean`, it will remove the program and object files, leaving you with your original, source-files-only working space. The target is phony in the sense that it does not actually make a file named `clean`. It is, however, often quite convenient.