# Quick start with C, `gcc`, and `gdb`

Begin with a simple integer temperature conversion program, without inputs or outputs, in a file named `conversion.c`.

conversion.c

```
1  int main(int argc, char **argv)
2  {
3          int fahrenheit, celsius;
4          celsius = -40;
5          fahrenheit = 9 * celsius / 5 + 32;
6          return 0;
7  }
```

There is one function, `main`; whenever any program is compiled and run, its `main` function is the starting point. Its arguments and return value are important, but not to this quick introduction. The first line of the body is a declaration, indicating that there shall be two variables, `fahrenheit` and `celsius`, both with type `int`, meaning that they can store integers. In the actual body of the function, first `celsius` is assigned a value, and then the appropriate temperature conversion is performed to assign the equivalent value to `fahrenheit` (up to the precision possible with integers, of course).

Compile the program with `gcc`.

As a typographical convention, `$` indicates a shell prompt, *`text like this`* is the user's input, and `this sort of text` is the output.

```
$ gcc -Wall -g -o conversion conversion.c
```

Modern compilers are also extremely advanced static analysis tools and are capable of detecting many bugs that will otherwise make you feel stupid after hours of hunting. You want the compiler to do your work for you, so the `-Wall` ("all warnings") option is essential. Because this program has no output, you will have to examine it in the debugger, `gdb`, in order to see what is going on. Passing `gcc` the `-g` option tells it to include debugging symbols in the compiled program so that `gdb` knows variable names and so forth. Finally, by tradition, `gcc` defaults to placing the compiled, executable program in a file named `a.out`; the option `-o conversion` specifies a more appropriate name.

Now try running the program.

Run `./conversion`, rather than plain `conversion`, to explicitly execute the file named `conversion` in the current working directory, which is named `.` ("dot"). This avoids the shell searching for your program on the `PATH`.

```
$ ./conversion
```

Naturally, it is silent. Bring it up in the debugger to see what is going on.

```
$ gdb conversion
(gdb) break main
Breakpoint 1 at 0x80483ba: file conversion.c, line 4.
(gdb) run
```

```
Starting program: /home/robert/CS2022/conversion

Breakpoint 1, main (argc=1, argv=0xbffff4e4) at conversion.c:4
4               celsius = -40;
(gdb) print celsius
$1 = 134513664
(gdb) print fahrenheit
$2 = 0
(gdb) next
5               fahrenheit = 9 * celsius / 5 + 32;
(gdb) print celsius
$3 = -40
(gdb) next
6               return 0;
(gdb) print fahrenheit
$4 = -40
(gdb) continue
Continuing.

Program exited normally.
(gdb) quit
```

Note that uninitialized variables have unpredictable values; in this run, `celsius` happened to be 134513664, and `fahrenheit` happened to be 0. Always make sure, as is the case in this example, that variables are assigned reasonable values before they are used!

In this case, `celsius` and `fahrenheit` were the same, because the two temperature scales cross at minus forty degrees. It is also possible to change a running program in the debugger. The `print` command accepts an arbitrary C expression, including assignments; we can use that to try other conversions without editing the code. (A better way, of course, would be to add some way of taking input; we will get there.)

The debugger also accepts non-ambigiuous prefixes of commands, for example `b` instead of `break`. Also, simply pressing enter will often repeat the previous command, which is particularly useful for commands like `next`.

```
$ gdb conversion
(gdb) b main
Breakpoint 1 at 0x80483ba: file conversion.c, line 4.
(gdb) r
Starting program: /home/robert/CS2022/conversion

Breakpoint 1, main (argc=1, argv=0xbffff4e4) at conversion.c:4
4               celsius = -40;
(gdb) n
5               fahrenheit = 9 * celsius / 5 + 32;
(gdb) p celsius
$1 = -40
(gdb) p celsius = 0
$2 = 0
(gdb) n
```

```
6               return 0;
(gdb) p fahrenheit
$3 = 32
(gdb) cont
Continuing.

Program exited normally.
(gdb) q
```

This first program uses only pure, ANSI C, without any libraries at all, even the C standard library. To produce output, however, we must introduce the standard library, and in particularly the very powerful function `printf` ("print formatted"). In order to access its declaration, you must direct the C preprocessor (more on that later) to include the header file `stdio.h` ("standard input and output").

As a powerful function, `printf` can also be a bit tricky. For example, it can take a variable number of arguments. The first and only required argument is a "format string"; most characters of the format string stand for themselves and are simply printed out. Thus, a more common first C program is the famous "hello world" program:

hello.c

```
1 #include <stdio.h>

2 int main(int argc, char **argv)
3 {
4         printf("Hello, world!\n");
5         return 0;
6 }
```

Notice the new first line to include `stdio.h`. The only tricky thing about this string is the sequence `\n`, which in C strings stands for the newline character.

```
$ gcc -Wall -g -o hello hello.c
$ ./hello
Hello, world!
```

In `printf` format strings, you can also specify additional arguments to the function that will be formatted and inserted in the appropriate place. For instance, if the format contains `%d`, an integer will be expected in the arguments to `printf` arguments, and it will be formatted in ordinary decimal and printed out. Using this, extend the temperature conversion program to be less mute.

conversion2.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;
}
```

And the result:

```
$ gcc -Wall -g -o conversion2 conversion2.c
$ ./conversion2
-40 Celsius is -40 Fahrenheit.
```