Chapter 41: Type Qualifiers

Section 41.1: Volatile variables

The `volatile` keyword tells the compiler that the value of the variable may change at any time as a result of external conditions, not only as a result of program control flow.

The compiler will not optimize anything that has to do with the volatile variable.

```
volatile int foo; /* Different ways to declare a volatile variable */
int volatile foo;

volatile uint8_t * pReg; /* Pointers to volatile variable */
uint8_t volatile * pReg;
```

There are two main reasons to uses volatile variables:

- To interface with hardware that has memory-mapped I/O registers.
- When using variables that are modified outside the program control flow (e.g., in an interrupt service routine)

Let's see this example:

```
int quit = false;

void main()
{
    ...
    while (!quit) {
      // Do something that does not modify the quit variable
    }
    ...
}

void interrupt_handler(void)
{
  quit = true;
}
```

The compiler is allowed to notice the while loop does not modify the `quit` variable and convert the loop to a endless `while (true)` loop. Even if the `quit` variable is set on the signal handler for `SIGINT` and `SIGTERM`, the compiler does not know that.

Declaring `quit` as `volatile` will tell the compiler to not optimize the loop and the problem will be solved.

The same problem happens when accessing hardware, as we see in this example:

```
uint8_t * pReg = (uint8_t *) 0x1717;

// Wait for register to become non-zero
while (*pReg == 0) { } // Do something else
```

The behavior of the optimizer is to read the variable's value once, there is no need to reread it, since the value will always be the same. So we end up with an infinite loop. To force the compiler to do what we want, we modify the declaration to:

```
uint8_t volatile * pReg = (uint8_t volatile *) 0x1717;
```

Section 41.2: Unmodifiable (const) variables

```
const int a = 0; /* This variable is "unmodifiable", the compiler
                    should throw an error when this variable is changed */
int b = 0; /* This variable is modifiable */

b += 10; /* Changes the value of 'b' */
a += 10; /* Throws a compiler error */
```

The `const` qualification only means that we don't have the right to change the data. It doesn't mean that the value cannot change behind our back.

```
_Bool doIt(double const* a) {
   double rememberA = *a;
   // do something long and complicated that calls other functions

   return rememberA == *a;
}
```

During the execution of the other calls `*a` might have changed, and so this function may return either **`false`** or **`true`**.

Warning

Variables with `const` qualification could still be changed using pointers:

```
const int a = 0;

int *a_ptr = (int*)&a; /* This conversion must be explicitly done with a cast */
*a_ptr += 10;          /* This has undefined behavior */

printf("a = %d\n", a); /* May print: "a = 10" */
```

But doing so is an error that leads to undefined behavior. The difficulty here is that this may behave as expected in simple examples as this, but then go wrong when the code grows.