



UNIVERSITY
of York

SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

Systems Programming for ARM Laboratory Scripts

Dr. Andy Pomfret

Laboratory 3

Type Qualifiers

3.1	Aims	35
3.2	The <code>volatile</code> Keyword	35
3.2.1	Unoptimised Behaviour	36
3.2.2	Optimisation Without <code>volatile</code>	36
3.2.3	Optimisation With <code>volatile</code>	37
3.3	The <code>const</code> Keyword	37
3.4	The <code>static</code> qualifier	38
3.4.1	Globals, <code>extern</code> and <code>static</code>	38
3.4.2	Static Locals	40
3.5	Summary and Conclusions	41

3.1 Aims

In this laboratory session, you will:

- Explore the `volatile`, `const` and `static` type qualifiers



Ensure that the ARM board is connected to your PC, and Launch μ Vision and CoolTerm as before.


Copy your project files from your `H:` drive, or wherever you put them last week, to your local `C:` drive, perhaps in `C:\tmp`. Remember that the debugger may not function correctly if you don't do this.

3.2 The `volatile` Keyword

The first part of this laboratory session is designed to help you to understand the relationship between the `volatile` keyword and the compiler.


The `volatile` keyword is a *type qualifier*: it does not in itself define a type, but it can be used as part of a type declaration to indicate to the compiler that a particular item of

data should be regarded as volatile. That is, that the data in question is subject to change between being written and being read.


 Use **Project**→**Open Project...** to open `Qualifiers\volatile\volatile.uvprojx`.

For this experiment, you will be using a *memory-mapped peripheral*. Don't worry exactly about how this works; essentially, there is a memory location whose contents always reflect the status of the pushbuttons on the development board. The memory location in question is `0x40021011`. Anything written to that location will be ignored, and whenever it is read the status of the buttons will be returned. You'll also notice that the project uses memory location `0x40020c15`, which is connected to the LEDs.


The interesting thing about this memory location is that it does not behave like RAM. Each time it is read, a different value may be returned – unlike RAM, where the value in a location will only change if the location is written to. This is something that the compiler needs to know about.

 The `#define` preprocessor directive declares that every time the word `buttons` appears in the code, it will be replaced by the given string. This is a useful technique for making code easier to read. We'll see more about preprocessor directives and macro replacements as the module progresses.

3.2.1 Unoptimised Behaviour


 Click the Target Settings button. Click the **C/C++ (AC6)** tab. In the **Language / Code Generation** panel, notice the drop-down menu labelled **Optimization**. It should say `-O0`, confirming that the compiler will not perform any optimisation. Click the **Cancel** button.


 Do not change any settings in the Target Settings window without knowing what you're doing. Doing so could stop the project from building, downloading, or running.

 Before you build and run the code, can you work out what it is supposed to do? Ask for help if you don't understand.

 Build and download the code, and verify that it behaves as you expect.

3.2.2 Optimisation Without `volatile`

 Click the Target Settings button again, and the **C/C++ (AC6)** tab. Pull down the **Optimization** box and select `-Ofast`. This will configure the compiler to apply maximum speed optimisation to the compiled code. Click the **OK** button. Build and download the code.

 What happens now when the code is run? Why do you think this might be?

3.2.3 Optimisation With `volatile`



Modify the definition of `buttons` to add the `volatile` type qualifier, so that

```
#define buttons (*(unsigned char *) (0x40021011))
```

becomes

```
#define buttons (*(unsigned char volatile *) (0x40021011))
```

Rebuild and download the project.



What behaviour do you observe now? Is this what you expected to see? Why?

3.3 The `const` Keyword

Now you will experiment with the `const` keyword, and see how it can help protect against programming errors. Again, the `const` keyword is a type qualifier; this time, it is effectively a promise that a variable will not be modified.



Use Project→Open Project... to open `Qualifiers\const\const.uvprojx`.

You should be able to see that the function `copyString(char *source, char *dest)` copies a string from one buffer to another, one character at a time, until a termination character is found.

- Build and run the code, and check that it behaves as you expect.
- Try swapping the parameters in the call to `copyString()`, so that `buffer` is copied to `message`. What happens? Why?



In general you can assume nothing about the data in an uninitialised array. If the buffer had contains random values, `copyString()` will blindly copy them to `message`, possibly overrunning the end of the array (remember, C has no bounds checking).



If you want to make sure that it is not possible to modify a variable by accident, declare it as `const`. Pointers to string literals should **always** use `const` to reflect the fact that the literal is immutable. This is actually required in C++, but is not required in C (even in recent variants) for reasons of backwards compatibility — the `const` keyword did not exist in the language originally.



Swap the parameters in the call to `copyString()` back to their original order. Check that it behaves as it did before. Now modify the code to use `const`:

- Add a `const` type qualifier to the `message` variable: `char const * message`
...
- Try to build the code. What happens?



You've declared the `message` variable as `const`, meaning that it is not to be modified. But when `copyString()` is called, it expects two non-`const` pointers. The compiler

notices this. Using the default settings a warning, rather than an error, is generated — but this highlights exactly why warnings should not be ignored.

The `copyString()` function must also be modified, so that it promises not to modify one of its arguments. This process of ensuring that `const` variables remain `const` is known as ‘const-correctness’.

- Add a `const` type qualifier to the `source` parameter declaration in `copyString()` so that it reads `char const * source`
- Try to build the code now. What happens? What happens if you swap the arguments in the call to `copyString()`, as before?

3.4 The `static` qualifier

Remember from the video that the `static` keyword has two distinct meanings. When applied to a *global variable*, or *function* its effect is to reduce its visibility so that no other files in the project can access it. When applied to a *local variable*, its effect is to increase its lifespan so that its value persists from one invocation of the function to another.

- ❗ Declaring a function as `static` is important if the function is only used locally; there is no need for the linker to see it, so it shouldn’t be visible. This compiler, with default settings, will warn about this: if you declare a function without the compiler having previously seen a prototype for it (which would typically be found in a header file) a warning will be generated. You can suppress the warning either by adding a prototype, or by declaring the function `static`.

3.4.1 Globals, `extern` and `static`

- ⚠ Global variables are usually a bad idea. It is a much better idea to pass data into a function, and to return a result, than to allow a function to read and write global variables. That’s because it’s impossible to control access to global variables – if one function can access them, all functions can access them. Small programming errors can lead to unexpected changes in the values of global variables, which can lead to seemingly unrelated problems in other functions.

However, occasionally in systems programming global variables are unavoidable. They are the only way to share information between processes in a system using a primitive operating system, for example. If a global variable is necessary, sometimes the `static` keyword can help to reduce these problems, providing the variable is only required within the current file.

- ⚙ Use Project→Open Project... to open `Qualifiers\static-global\static-global.uvprojx`.

These simple exercises will demonstrate how the `static` keyword restricts visibility, and how the `extern` keyword separates declaration from definition.

- Note that there are two source files in this project: `main.c` and `count.c`. Both of these `#include` a header file, `count.h`.
- In `count.h`, a global variable `counter` is declared and defined. Also, a prototype for a function called `count()` is provided.
- In `count.c`, the function `count()` is defined. It simply increases the value of the global `counter` variable.

Try to build the code. You should find that you get an error, and that the error occurs *after* `main.c` and `count.c` have been successfully compiled. This is a *linker* error.

- ❗ The linker gives an error saying that the symbol `counter` is defined in two places: `count.o` and `main.o`. To understand why, remember the three steps in the compilation process.
- First, the preprocessor takes the contents of `count.h`, along with all other included header files, and effectively pastes it into `main.c` and `count.c`. Now both source files contain the line

```
unsigned int counter;
```

from `count.h`.
 - Second, the compiler compiles the two source files independently of each other. Each one compiles fine, but each one declares and defines a global variable called `counter`. These are two different variables, both visible globally, and both with the same name.
 - Finally, the linker tries to sort out references to global symbols within each compiled object file. It sees two global variables, both called `counter`, and reports an error.
- ❗ Note that the compiler notices that something is amiss. It generates warnings from `count.h`, twice (once when it's included in `main.c`, once when it's included in `count.c`). These warnings are a bit like the ones you get for a non-static function with no prototype, and they essentially mean "you've declared a non-static global variable here, are you sure that's what you want?"

3.4.1.1 Using `extern`

To fix this problem, the `extern` keyword can be used. When applied to a global variable, `extern` tells the compiler to *declare* the variable but not to *define* it. This is analogous to a function prototype, which is a declaration (it tells the compiler that the function exists) but not a definition (it doesn't say what the function does, and the compiler doesn't generate the function at that point).



Change the declaration of `counter` in `count.h` so it reads

```
extern unsigned int counter;
```

Try to build the code now. What happens? You should find that you still get a linker error, but it's changed – it now says that the symbol `counter` is undefined. The compiler warnings are gone, too.



We now have the opposite problem: using the `extern` keyword means that, after the preprocessor has done its job, both `main.c` and `count.c` *declare* the variable, but neither of them *defines* it. When the linker looks for the variable, it doesn't exist – the compiler hasn't created space for it anywhere. Again, this is analogous to function prototypes; if every file contains a prototype for a particular function, but the function is never actually defined, the linker will complain in the same way.



To fix this, one (and *only* one) source file must define the variable. Add the line

```
unsigned int counter;
```

to `count.c`, above the function declaration. Build and run the code. Does it do what you expect?

3.4.1.2 Using `static`



Remove the line you've just added to `count.c`. Change the declaration of `counter` in `count.h` to read

```
static unsigned int counter;
```

Build and run the code. What happens?

It's important that you understand what is happening here. Use your knowledge of the three build steps – preprocessing, compilation and linking – to work out what the compiler and linker are 'seeing' in this case. If you don't understand, please ask for help.

3.4.2 Static Locals

Used within a function, the `static` keyword indicates that a variable is to be stored *statically* rather than *dynamically*. This means that dedicated storage will be allocated for the variable, instead of it residing on the stack. The important side-effect of this is that the contents of the variable will be preserved when the function ends, and will still be present next time the function is called.



Static local variables are also usually a bad idea, for a couple of reasons:

- When you're reading someone else's C code, you expect local variables to be re-initialised whenever a function is called. Static variables don't behave like that, so they can make code harder to read and understand.

- A function that uses static variables is not *re-entrant*: if you're using an operating system, and two processes try to use the function at the same time, the function will not behave as you expect.



Use Project→Open Project... to open `Qualifiers\static-local\static-local.uvprojx`.

- The code in this project is very simple. Check that you understand what it will do, then build and run it.
- Now change the declaration of `counter` from

```
unsigned int counter = 0;
```

to

```
static unsigned int counter = 0;
```

and rebuild and run the code. What happens? Why?



Recall that initialisation takes place at the time of variable creation. When the local variable is declared **static** it has program lifetime, so it is only created (and therefore only initialised) once. This highlights the difference between initialisation and assignment.



Once you've closed μ Vision, copy your project files back to a network drive for next time.

3.5 Summary and Conclusions

By now, you should:

- Be more familiar with the operation of the type qualifiers `const`, `volatile` and `static`;
- Understand more about why an appreciation of the build process is important when working in C.

If you have any questions about what you have learned, please ask.