UNIVERSITY of York

SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

Systems Programming for ARM Laboratory Scripts

Dr. Andy Pomfret

Laboratory 2

C programming revision

2.1 Aims

In this laboratory session, you will:

- Revise some basic but important C programming concepts;
- Get more experience with the programming environment.

It is assumed that you are familiar with functions; parameters and return values; loops using the `for`, `while` and `do while` constructs;

Ensure that the ARM board is connected to your PC, and Launch µVision 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.

2.2 Declaration, Definition, Initialisation and Assignment

It will be important, later in this module, to differentiate between the *declaration* of a variable and the *definition* of that variable, and between the *initialisation* of a variable and *assignment* of a value to it.

There are no exercises to complete here but please make sure you read this section and understand it before you move on.

First, a few definitions.

- **Declaration** is the process of telling the compiler that something exists, and associating a name with that thing.
- **Definition** is the process of requesting that space be allocated to store something. Declaration and definition are not the same thing. They are almost always performed together when it comes to variables, but may be separated in certain circumstances.
- **Initialisation** is the process of providing a variable with an initial value. This value is given to the variable when it is created; exactly when this takes place depends on the nature of the variable, and is something we'll look at another time.
- **Assignment** is the process of changing the value stored in a variable, after the time of its creation. Assignment and initialisation are not the same thing.

Some examples:

```
int x;      // Declares and defines x
int y = 3; // Declares, defines and initialises y
x = 3;      // Assigns to x
```

These example variables both use the type `int`. There are quite a few data types in C, and it's worth investigating them and how they behave and interact.

2.3 Data Types, Type Safety and Casting

Simple numeric data types in C fall into two categories: integer and floating-point types.

2.3.1 Integer Types

Table 2.1 lists the numeric integer types available on this platform for your reference.

Name	Size	Minimum	Maximum	`printf` specifier
`signed char`	8-bit	-2^7	2^7-1	`hhd`
`unsigned char`	8-bit	0	2^8-1	`hhu`
`short`	16-bit	-2^{15}	$2^{15}-1$	`hd`
`unsigned short`	16-bit	0	$2^{16}-1$	`hu`
`int`	32-bit	-2^{31}	$2^{31}-1$	`d` or `ld`
`unsigned int`	32-bit	0	$2^{32}-1$	`u` or `lu`
`long`	32-bit	-2^{31}	$2^{31}-1$	`d` or `ld`
`unsigned long`	32-bit	0	$2^{32}-1$	`u` or `lu`
`long long`	64-bit	-2^{63}	$2^{63}-1$	`lld`
`unsigned long long`	64-bit	0	$2^{64}-1$	`llu`

Table 2.1: Numeric integer data types in C on a 32-bit platform

ⓘ There is also the `char` data type, which the standard does not specify as being either signed or unsigned, and which is intended specifically for holding ASCII character data.

ⓘ Note that `int` is usually a synonym for `long` on a 32-bit platform, but a synonym for `short` on a 16-bit platform. It can even be a synonym for `long long` on some 64-bit platforms. Other differences between platforms exist too. The standard only specifies an ordering for the sizes of the types, and minimum sizes for most. We will look at a more modern solution for this problem another time.

2.3.2 Floating-Point Types

Floating-point data types provide a much wider dynamic range than integer types, and can be useful things. There are three that are available:

- `double` is a 64-bit floating point type, on almost every platform implemented using the IEEE 754 specification for double-precision binary floating-point values (11-bit exponent, 53-bit mantissa). The format specifier for `printf()` is `%lf`.
- `float` is a 32-bit floating point type, on almost every platform implemented using the IEEE 754 specification for single-precision binary floating-point values (8-bit exponent, 24-bit mantissa). There is no format specifier for `printf()`, but you can use `%f` and the value will be converted to a `double` when it's passed to `printf()`.

- `long double` is a floating point number of unspecified size and precision, guaranteed to be at least as big as a `double`. In practice on small embedded systems this is almost always equivalent to a codedouble. For what it's worth, the format specifier for `printf()` is also `%lf`.

⚠ Floating-point values can appear attractive, due to their wide dynamic range. However, they have a number of disadvantages compared with integer types:

- They can suffer from *catastrophic cancellation* and other types of rounding error propagation, which can lead to bugs that are difficult to identify.
- They require considerably more processing power, especially on microcontrollers, many of which do not have dedicated floating-point hardware. The STM32F407VG, having a Cortex-M4F core, *does* have an FPU but it is single-precision (meaning `float` is fast but `double` is not), and its use consumes more power.

On the whole it is a good rule of thumb to **avoid using floating-point types on embedded systems** unless you have a very good reason to use them.

It's time for an exercise. The `sizeof` operator in C can be used to find the size of any given thing, at compile time, in bytes (with caveats). The type of the value returned by `sizeof` is effectively `unsigned int`. Use it to confirm that the integer and floating-point types are the size I've said they are! There's no need to check both the signed and unsigned versions of each integer type. Start by using Project→Open Project... to open `Programming\types\types.uvprojx` which contains an example.

2.3.3 Casting and Promotion

C has a reasonably strong type system. Generally speaking, when assigning a value to a variable, the value and the variable have to share a type.

In some cases, the compiler will allow you to assign a value of one type to a variable of another without complaining. For example, a value of type `short` can be assigned to a variable of type `long`; the compiler is aware that this conversion can always be performed with no loss of information, and will automatically insert the correct code for the conversion. Conversions in the other direction, however, will generally result in warnings.

Add the code

```
int x = 42;
int y;

y = x;
```

to the existing project.

This code doesn't do anything useful at all. But I don't intend you to run it — for the moment, all of the useful learning will come via the build process. So first of all, build it, and check that it builds without warnings.

Change the types of x and y, and rebuild the code. Try different combinations of integer types, both signed and unsigned, and see what happens.

You should observe the following behaviour:

- If the *signedness* of x and y is the same, and y has at least the same *width* as x (width is the number of bits in the type), no warning is given because all of the possible values that x can hold can also be represented by y. The compiler silently generates code to perform the conversion.
- If y has a *greater width* than x, and x is unsigned but y is signed, no warning is given. Again this is because the range of values that y can hold completely encompasses the range of values that x can hold, and the compiler silently generates code to perform the conversion.
- If x and y are the same with and their signedness differs, **or** if y is wider than x but y is unsigned while x is signed, you will see a warning such as

  ```
  src/main.c(11): warning: implicit conversion changes
     signedness: 'int' to 'unsigned long long' [-Wsign-
     conversion]
          y = x;
            ~ ^
  ```

  This indicates that the target variable is being assigned a value that it might not be able to represent, and that the likely manifestation of the problem is that it will be assigned an erroneous value that differs in sign from the original value.
- If x is wider than y, then regardless of the signedness of either variable, the range of values that y can hold is smaller than the range that x can hold. The result is a warning like

  ```
  src/main.c(11): warning: implicit conversion loses integer
     precision: 'unsigned long' to 'unsigned short' [-
     Wimplicit-int-conversion]
          y = x;
            ~ ^
  ```

The silent assignment of a value to a wider variable is known as *promotion*. Integer promotion takes place whenever an arithmetic operation involves values of different types. It is rarely an important consideration these days, especially when working on 32-bit or 64-bit platforms. But knowing that it happens could be important if you were working on an 8-bit platform, for example.

It is possible to suppress the warnings you've just seen by using a *cast*.

2.3.3.1 Casting

A *type cast*, or just a *cast*, is an explicit request for a type conversion. The syntax is quite simple: place the desired type in parentheses in front of the value to be converted. For example,

```
int x = 42;
unsigned int y = 42;

y = (unsigned int) x;
```

Try replicating this example, and check that it builds without warnings.

Unfortunately, **casting is not magic**. If the target variable has a type that is not capable of holding the value it's assigned, you will still have a problem.

Change the example so that the constant is `-42` instead of `42`, and then add a line that prints the values of `x` and `y`. (Note the format specifiers of `%d` and `%u` for `int` and `unsigned int`, respectively.) What do you observe?

Try another example, changing `x` to be an `int` and `y` to be a `short`, and using the constant `42000`. The format specifiers are `%d` and `%hd` respectively. Can you explain what you see?

A cast, usually, is simply an instruction to the compiler to ignore any concerns it may have about a type conversion. When you use a cast, you are telling the compiler that you know best: so **when you use a cast, you're on your own**. Do not use a cast unless you know that you need one — and even then, consider if there is a better way to work (e.g. by changing the types of one or more variables). Finally, bear in mind that if you're having problems with your code, anywhere you've used a type cast is a good place to start looking! This will particularly apply when we start casting pointer types later in the module.

2.3.4 Literals

A *literal* is any hard-coded constant with a fixed value that's known at compile time. In C, by default, any numeric literal *without* a decimal point is assigned the type `int`. Any numeric literal *with* a decimal point is assigned the type `double`.

You can add suffixes to numeric literals to change their types. This can be useful if you wish to avoid unnecessary promotion of floating-point variables, or spurious warnings about fixed point sign changes from the compiler, as well as to define integer literals larger than the size of an `int`.

2.3.4.1 Integer literals

The integer literal suffixes are:

- `U` for `unsigned int`
- `L` for `long`
- `UL` for `unsigned long`
- `LL` for `long long`
- `ULL` for `unsigned long long`

In general, the compiler is smart enough to avoid warning you about a loss of integer precision when you use a standard integer literal to initialise or assign to a variable narrower than an `int`. For example, as you've seen,

```
short x = 42;
```

will compile just fine because although the literal is technically an `int` the compiler knows its value and knows that it will fit fine into the range of a `short`. However, sometimes it can be important to specify the size of a literal, and so it is often considered good practice to do so always.

Add the code

```
unsigned int mask = (0xBA) << 8;
printf("mask: 0x%08X\r\n", mask);
```

to the project. Build and run the code to see what it does.

The bit shift operators » and « are used to shift a number right or left by the given number of bits. This is useful for fast integer multiplication and division by powers of two, but most commonly it's used when creating patterns of bits to use as masks or for configuration of peripherals.

You should find that the code above prints out the value `0x0000BA00` which is the result of shifting the value `0xBA` 8 bits to the left.

Change the number of bits that the pattern is shifted from 8 to 24, and build the code again.

You should see two warnings from this line now:

```
src/main.c(15): warning: signed shift result (0xBA000000) sets
   the sign bit of the shift expression's type ('int') and
   becomes negative [-Wshift-sign-overflow]
        unsigned int mask = (0xBA) << 24;
                            ~~~~~~ ^  ~~
src/main.c(15): warning: implicit conversion changes signedness:
    'int' to 'unsigned int' [-Wsign-conversion]
        unsigned int mask = (0xBA) << 24;
                     ~~~~   ~~~~~~~^~~~~
```

The second of these warnings is one you've seen before: it's warning about the implicit conversion of an `int` to an `unsigned int`. But the first is specific to these kinds of bit-shift operations: it's generated before the conversion to `unsigned int` is even considered, and is generated because the result of the bit shift is considered to be an `int` because that's the type of the value being shifted, and the result `0xBA000000` has a different sign from the input `0x000000BA`.

If you run the code you'll find that it works just fine; the value `0xBA000000` represents a negative value as an `int`, but then the conversion to `unsigned int` shows that sometimes two wrongs can make a right, and the value is interpreted correctly. But we can do better.

Change the code so that the literal is `0xBAUL`. The `UL` suffix will change the type of the literal to `unsigned long`, which in turn will change the type of the shift result and remove the warning about the sign of that result, as well as removing the type conversion when the value is assigned to the variable. Build and run the code: check that the warnings are gone, and that the program runs as expected.

2.3.4.2 Floating-point literals

For floating-point numbers you can add the suffix `f` to indicate a single-precision floating-point literal. This can result in significantly faster code, especially on this platform.

Use Project→Open Project... to open `Programming\float\float.uvprojx`.

Take a look at this code. Pretty simple, right? It should just increment a floating-point value by 0.3 and print out its new value, repeatedly.

Build the code, but don't run it yet. Take a look in the build log at the bottom of the screen. You should see the following at the bottom of the log:

```
".\Objects\float.axf" - 0 Error(s), 2 Warning(s).
```

Let's go and look at those warnings. Scroll up, and you'll find:

```
src/main.c(11): warning: implicit conversion when assigning
   computation result loses floating-point precision: 'double' to
    'float' [-Wimplicit-float-conversion]
                t += 0.3;
                  ~~ ^~~
src/main.c(12): warning: implicit conversion increases floating-
   point precision: 'float' to 'double' [-Wdouble-promotion]
                printf("t = %f\r\n", t);
                ~~~~~~                ^
```

These two warnings are telling you different things, but they are both related. In both cases, the warning means that implicit conversion has taken place between `float` and `double`, and that you should make sure that that's what you wanted.

2.3.4.3 Implicit float conversion

The first of the two warnings is generated because the floating-point literal `0.3` is assumed to have type `double`, which causes the value of the variable `t` to be promoted to `double` for the addition. The result of this addition is of type `double`, but the destination for the result is the variable `t` which is a `float`, and it's this conversion of the result that generates the warning.

Let's see what effect this conversion to and from `double` is having on the code.

Place a breakpoint on the `t += 0.3;` line and another on the `printf()` line just below it. This will help you to identify them in the disassembly listing. Now start the debugger, and run to the first breakpoint.

Take a look in the disassembly window. The program will be paused at the start of the set of instructions that implement the `t += 0.3` operation. Try to find the end of that operation, which will be marked by the next breakpoint. (Expanding the disassembly window downwards may help with this.) You should find that that one simple line of C translates to 11 machine code instructions:

```
0x08002484 9802      LDR           r0,[sp,#0x08]
0x08002486 F000FE7D  BL.W          __aeabi_f2d (0x08003184)
0x0800248A EC410B10  VMOV          d0,r0,r1
0x0800248E ED9F1B10  VLDR          d1,[pc,#0x40]
0x08002492 EC532B11  VMOV          r2,r3,d1
0x08002496 EC510B10  VMOV          r0,r1,d0
0x0800249A F000FC8D  BL.W          __aeabi_dadd (0x08002DB8)
0x0800249E EC410B10  VMOV          d0,r0,r1
0x080024A2 EC510B10  VMOV          r0,r1,d0
0x080024A6 F000FC55  BL.W          __aeabi_d2f (0x08002D54)
0x080024AA 9002      STR           r0,[sp,#0x08]
```

We'll be looking at ARM assembly language soon, but for now just note that this is even worse than it looks because three of these 11 lines are function calls (the lines containing `BL.W`). The first converts the value of `t` to a `double`; the second performs a double-precision add; and the third converts the result back to a `float`. As you can imagine, this is not efficient.

Close the debugger, and add the `f` suffix to the literal `0.3` so the line reads `t += 0.3f;`. Recompile the code.

Note how the first of the two warnings has gone. Reopen the debugger and run to the first breakpoint. This time you should find that the same line of code translates to just four instructions, none of which is a function call:

```
0x08002480 ED9D0A02  VLDR          s0,[sp,#0x08]
0x08002484 ED9F1A0A  VLDR          s2,[pc,#0x28]
0x08002488 EE300A01  VADD.F32      s0,s0,s2
0x0800248C ED8D0A02  VSTR          s0,[sp,#0x08]
```

All four of these lines are native floating-point instructions. Bear in mind that this result was obtained with compiler optimisation disabled, so there is probably more performance still to be found here. But by understanding the nature of the hardware, and the behaviour of literals, a tiny change to the C code can make a huge difference to the compiled result, without any change whatsoever to the behaviour of the code.

2.3.4.4 Double promotion

The remaining warning in the code occurs when we attempt to print out the value of the variable `t`. This warning is telling us that `printf()` is expecting a **`double`** and we're passing a **`float`**, so the value will be converted.

There's nothing much we can do about this. There is no way to force `printf()` to accept a **`float`**, so the promotion to **`double`** is inevitable. It will involve a function call, and will be reasonably computationally expensive, but there is no other way to achieve the desired outcome.

However, just because there's no way to fix the problem doesn't mean there's no way to address the warning. The warning exists because *there is nothing in the code to make it clear that the promotion is expected*. We can remove the warning by indicating explicitly to the compiler that we want to convert the value to a **`double`**.

Insert a type cast to convert `t` to a **`double`** explicitly. The line should read

```
printf("t = %f\r\n", (double)t);
```

Rebuild the code and verify that it now builds without warnings.

Warnings

In general you should go to whatever lengths you need to go to to ensure that your code **builds without warnings**. There are various approaches to this, all of which might be appropriate at different times. The two we've just seen are to *modify the code* to change its behaviour in response to the warning, and to *make an operation explicit* to suppress the warning without changing the behaviour of the code. These two are the approaches you should favour wherever possible. We'll look at some others later in the module.

ARM Compiler 6, which is based on clang, will issue the warnings you have seen when the `-Wall` flag ("all warnings") is given to the compiler, as it is in all of these projects. But not all compilers will do so. **Do not rely on the compiler to check your code for you**.

Run the code, if you haven't already, and watch the values increase. What do you notice? Why is this happening?

2.4 Pointers and Pointer Arithmetic

In this short exercise we'll take a look at what happens when a pointer is modified using an arithmetic operation.

There is a `printf()` format specifier for printing out the value of a pointer: `%p`. Although the specifier is standardised, its behaviour is not exactly consistent. The C99 standard says in §7.19.6.1 ¶8:

> The value of the pointer is converted to a sequence of printing characters, *in an implementation-defined manner*.

Technically this format specifier expects a parameter of type `void *`, but don't worry about that for the moment — we'll look more at `void` pointers, and their uses and behaviour, another time.

The implementation-defined nature of the `%p` format specifier means that you can't rely on the format of what is printed. All implementations I'm aware of will print the address in hexadecimal, though that is not required. Some will add a leading `0x` to the value to indicate the base, others won't. Some will print in upper case, others in lower case.

This lack of consistency is not really a problem though, because it's very hard to think of a situation in which printing out the memory address of something would be useful in production code. This is intended as a debugging feature.

Use Project→Open Project... to open `Programming\pointers\pointers.uvprojx`.

Add a line of code to print out the address of the variable `i` (`&i`) in your code. Either in the same line, or in another line, print out the value of `(&i)+1`.

To suppress the warnings, it's easiest to perform an explicit cast to `void *` (in other words, use `(void *)&t` and `(void *)((&t)+1)`).

Before you build and run the code, what do you expect to see?

Now build and run the code, and check to see if your expectation was correct.

Note how the value of `(&t)+1` is *not* one greater than the value of `&t`. That's because when you add an offset to a pointer, the pointer is increased by the number of bytes used by the target of the pointer.

Change the type of `t` from `int` to `long long`. Build and run the code, and check that the difference between `&t` and `(&t)+1` is now 8 bytes, reflecting the size of a `long long`.

2.4.1 Functions and pointers

It is reasonably common practice to pass pointers into functions, rather than passing the data itself, for a number of reasons:

- If an item of data is (or could be) large, such as a `struct` for example, passing it directly into a function could be computationally expensive. That's because every variable that's passed into a function is actually copied, and the function receives a copy. Avoiding this copy operation can increase speed and reduce memory usage.
- A function can return a value, but only *one* value. It can be very useful to allow a function to "return" more than one value by passing it pointers to variables that it can modify directly in lieu of a return.

Add a function called `square()` that takes a pointer to a `long long` and squares the value that it points to. The function should return `void`. Using the `static` keyword will suppress a compiler warning (again, we'll look at this another time).

Call the function from `main()`, passing it a pointer to the variable `i`, and print out the value of `i` before and after the call.

Build and test your code. Does it behave as you expect?

2.5 Arrays, Strings and Pointers

I'm going to assume that you're familiar with the basics of arrays in C: they're zero-indexed, statically sized, and elements are accessed using square brackets.

In this section we're going to review the links between arrays, strings and pointers.

2.5.1 Arrays

There are three possibilities for sizing and initialising an array:

- You can specify a size, but no initialisation data. This results in an array with **uninitialised** elements. They aren't zero, their contents are simply unknown and could be anything.
- You can specify no size, and provide comma-separated initialisation data in curly braces. The compiler will infer the array size from the length of the list.
- You can give both a size and an initialisation list, as in this case. The list will be used to initialise the array values until it is exhausted, and the remainder will be initialised to zero. This is commonly used to initialise an entire array to zero, by writing e.g.

  ```
  int x[100] = {0};
  ```

which initialises the first element to zero explicitly and the remainder to zero implicitly.

⚠ Beware: if you do not initialise an array when you declare it, then the values in that array are essentially random and cannot be trusted until you've changed them. The actual mechanism for this is that the memory occupied by your array is likely to have been used for something else in the past, and whatever was stored there previously will still be in that bit of memory when it's allocated for use by your array. Often, the "debug" settings for a compiler will generate code that initialises arrays to zero, which can mask any problems related to this oversight. In short: **initialise your arrays**! Most commonly this will mean initialising to zero, unless you have reason to initialise to some other value or values.

2.5.1.1 Indirection and pointer arithmetic

Arrays in C behave in very similar ways to pointers. In fact an array, and a pointer to the first element in the array, are basically the same thing for most purposes. This is due to a language feature known as *pointer decay*: essentially, as soon as you treat an array as if it was a pointer, it becomes one. The array "decays" to a pointer.

It's possible to use the indirection operator (*) with arrays, just as you would with a pointer.

Use Project→Open Project... to open `Programming\arrays\arrays.uvprojx`.

Look at the code and work out what it's going to do, then build it, and optionally run it to check that you were right!

Now change the line that says

```
numbers[0] = 11;
```

to read

```
*numbers = 11;
```

This works even though the variable `numbers` is an array, not a pointer, because as soon as we treat it as a pointer, it becomes one. Specifically, it becomes a pointer to the first element of the array, so we would expect this to overwrite the first element in the array, exactly the same as the line it replaced.

Build and run the code, and check that nothing has changed in the program's behaviour.

Now try replacing the line that says

```
numbers[3] = 42;
```

to read

```
*(numbers + 3) = 42;
```

and check once again that the behaviour of the program has not changed.

2.5.1.2 Square brackets

The square bracket notation for array indexing doesn't just work with arrays, it also works with pointers.

Declare a pointer and make it point to the first element of the `numbers` array. You could do this in either of two ways. You could rely on pointer decay:

```
int *pointer = numbers;
```

or you could explicitly find the address at which the first array element is stored:

```
int *pointer = &(numbers[0]);
```

Either way will work and both will have the same effect.

Now write another loop, just like the one you already have, to print out values using `pointer` as if it was an array. Something like this:

```
for (int i = 0; i < 5; ++i) {
  printf("pointer[%d] = %d\r\n", i, pointer[i]);
}
```

Build and run this code, and see what happens.

Square brackets for array indexing are an example of "syntactic sugar": they do not do anything that you couldn't do another way, but they can make your code more readable. In general

```
a[b]
```

is exactly synonymous with

```
*(a + b)
```

and either syntax can be used interchangeably.

It is sensible to use the square bracket syntax whenever you are accessing data *as if it was in an array*, whether or not it actually is.

Finally, add code to print the size of the `pointer` and `numbers` variables (using the `sizeof` operator). Build your code and see what happens.

This is almost the only time when the behaviour of an array differs from that of its decayed pointer equivalent. My strong advice would be **never** to use `sizeof` to determine the size of an array — doing so is dangerous because small changes to the code could lead to you inadvertently finding the size of a pointer instead, and the introduction of bugs. The only context in which `sizeof` can determine the size of an array anyway is when its size is known locally at compile-time, which means you know it too!

2.5.1.3 Arrays and functions

Whenever an array is passed to a function, it decays to a pointer.

This has several implications:

- Passing arrays into functions is **fast**, independent of the size of the array, because the array itself is not copied.
- Changes made to arrays inside functions **are visible in the calling scope**. As soon as the function dereferences the pointer it has been passed (whether by using the indirection operator or by using square brackets) it is accessing the *original array*, not a copy of it.
- Functions have no hope at all of working out how large an array is, because all they're given is a pointer to the first element.

There are two ways of declaring a function that accepts an array parameter. The following two lines of code give identical results:

```c
void function(int *array)
void function(int array[])
```

Whichever you use, the `array` parameter **will be a pointer**. From the C99 spec, §6.7.5.3, ¶7:

> A declaration of a parameter as "array of *type*" shall be adjusted to "qualified pointer to *type*", where the type qualifiers (if any) are those specified within the `[` and `]` of the array type derivation.

This, in turn, is effectively due to *decay* again, as summarised in §6.3.2.1, ¶3:

> Except when it is the operand of the `sizeof` operator or the unary `&` operator, or is a string literal used to initialize an array, an expression that has type "array of *type*" is converted to an expression with type "pointer to *type*" that points to the initial element of the array object and is not an lvalue. If the array object has register storage class, the behavior is undefined.

(An lvalue is anything which has, or could have, a defined address in memory.)

My preference is therefore for the **first** form because it's more explicitly obvious that it *is* a pointer.

Write a function called `processArray()`. It should:

- Take an array of `int` values as its only parameter
- Change the value of the 3rd array value to equal 10

Remember to place the function above `main()`, or else write a prototype for the function and place that above `main()` and the function itself below. Call the function from within `main()`, and print out the value of the 3rd array element before and after the function call.

Build and test your code. Does it behave as you expect?

2.5.2 Strings

There's no such thing as a string in C, really — at least, there's no actual string *type*. Standard C strings are essentially just arrays of ASCII characters, which in turn are just 8-bit (technically 7-bit) numbers. And because arrays and pointers are so closely related, you can also consider a string to be a pointer to the first character of the string.

Functions that accept strings as parameters will receive a pointer to the first character of the string, whether or not the string was previously considered to be an array. But these functions will need to know how long the strings are. For this, the convention is that each string ends with a *termination character*.

Termination characters

Not all ASCII character codes translate to printable characters. Codes 32 to 126 contain the letters, numbers and punctuation. Some other codes have printable meanings, including 10 (`'\r'`) and 13 (`'\n'`), which are the "carriage return" and "new line" codes respectively. But ASCII code 0 has a special meaning: **it indicates the end of a string**.

Note that ASCII 0 is not the same as the printable character `'0'`, which has the value 48. The termination character is stored numerically as a zero, and can be generated using either the integer literal `0` or the character literal `'\0'`.

The requirement for a termination character means that the space required for a string is always one byte larger than the number of characters in the string.

If a string doesn't have a termination character, for any reason, functions that rely on the termination character to calculate string length will misbehave. Over the years, this has been a common source of bugs and security vulnerabilities in code. We'll investigate this more closely another time.

2.5.2.1 String literals

Just like any other literal value, a *string literal* is a string value that's known at compile time. You have been using them as the first parameter to every `printf()` statement: a string literal is specified by using a pair of double-quotes, with the desired string enclosed between them.

String literals **include the termination character**, so for example the literal

```
"hello"
```

is equivalent to the characters `'h'`, `'e'`, `'l'`, `'l'`, `'o'`, `'\0'`.

Use Project→Open Project... to open `Programming\strings\strings.uvprojx`.

This code creates two strings, both containing the same text. The first is

```
char array[] = "Hello!";
```

which is an ordinary array, initialised using a string literal, just like the ones you've seen before. The second is

```
char * literal = "Hello!";
```

which is a *pointer* to the literal itself. Remember that arrays and pointers behave almost identically, so we can treat these two variables basically the same as each other.

Build and run the code. You should see that both strings are printed out, as well as the addresses at which those strings are stored.

You should see that the array is stored at an address like `0x2000XXXX`, whereas the literal is stored at an address like `0x0800XXXX`. Whereas addresses starting `0x2000` are in RAM (ordinary writeable memory), addresses starting `0x0800` are in the FLASH ROM (read-only memory) on the STM32F407. In other words, this string literal is stored somewhere within your code itself.

This makes sense because for the code to run correctly when power is first applied to the board, the string must exist in some memory that isn't erased when power is lost.

Try adding the lines

```
array[1] = 'u';
literal[1] = 'u';
```

after the variable declarations.

Build and run the code. What happens?

You will notice that while the modification of one character of the `array` variable was successful, the modification of one character of the `literal` variable was not. This is unsurprising given that the literal is stored in read-only memory, but you might expect some kind of compiler warning. And yet, nothing. We'll revisit this. But for now, remember: like any other literals, **string literals are immutable**.

It's time for some more experiments that reveal interesting information about how strings are stored.

First, remove the lines that change individual characters in the strings, to undo the changes you made for the last experiment.

Now add the line

```
char * literal2 = "Hello!";
```

to the existing set of declarations. Add a line to print its text, and a line to print the address at which it is stored.

? Before you run this code, where do you think this will be stored?

String pooling

You should see that the two pointers `literal` and `literal2` have the *same value*. They point to the same place, and there is actually only one string literal. Indeed, across all four of your variable declarations, only one string literal is used. This is due to **string pooling**.

String pooling is a technique that compilers use to reduce the size of generated code, by not storing duplicate information. When string pooling is used, as it always is on modern compilers, any given string literal is only stored once, and all pointers to that string will point to the same place.

Compilers are even allowed to use pooling for *overlapping* literals. For example the string literals "hello, world" and "world" could conceivably occupy the same space, sharing a termination character.

2.5.2.2 Strings and functions

We've now seen that a string in C is just a pointer to the first character in the string. You can use an array to hold a string, but as arrays decay to pointers so readily this ends up being the same thing. String lengths are not usually stored or passed around because strings have termination characters, so their ends can be found.

This implies a few things about strings and functions:

- Just like arrays, strings are not copied when they're passed to functions. A copy of the *pointer* is passed, but the underlying string data is not copied.
- If you write a function that processes a string, you should use the termination character to work out when to stop processing rather than requiring the string length as a separate parameter.
- It is **very difficult** to write a function that creates a new string and returns it, for a number of reasons (please ask for more information if you're interested). As a result, the standard string processing functions that do things like copying a string from one place to another usually accept a pointer to a *place to put the result* known as a "buffer" as a parameter, along with an additional parameter indicating how big that storage space is. For an example, see `strncpy()` (Appendix 2A.2).

Use Project→Open Project... to open `Programming\wordcount\wordcount.uvprojx`.

Your task is to complete the provided function `wordcount()`. (Don't worry about the function of the `static` keyword for now if you're not familiar with it.) This function takes a string as its only parameter, and should return an integer to indicate how many words are in the string. A word is defined as one or more consecutive alphanumeric characters (0-9, A-Z, a-z).

This is a surprisingly complex task, but you should have the skills you need to complete it. Some hints you might need:

- You can iterate over the characters in the string using a `while` loop. You can either create a counter and use it to index into the string like an array, or you can just keep increasing the `string` variable as you loop.
- For each character in the string, you can check if it's a letter (or number) by looking at its value. For example

  ```
  if (character >= 'A' && character <= 'Z')
  ```

  will check to see if the character is a capital letter.
- One way to count the words is to take a note of each time a word starts or ends, as you loop. Each time a word starts, you can increase the word count by one.

2.6 Summary and Conclusions

By now, you should:

- Recall the basics of pointer operation and manipulation in C
- Understand the "decay" of arrays to pointers
- Know about strings and string literals

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

Remember to copy your project files to a network drive before you log off.

Appendix 2A Standard string functions

The C standard library contains a number of functions that can be helpful when manipulating strings. Unfortunately they all seem to work in different ways! That's particularly true when it comes to their handling of termination characters. The following is a quick guide to the most useful of them, with simple examples.

Functions to avoid

⚠ Many of the functions listed here are modern variants of legacy string functions. The legacy functions should be considered unsafe, and should never be used.

2A.1 Finding string length: `strlen`

The `strlen()` function is pretty simple: it returns the length of the string it's given. The prototype in `string.h` is

```
size_t strlen(const char *str);
```

The `size_t` type is an unsigned integer type which is capable of storing "the maximum size of a theoretically possible object of any type". In practice it's almost always the same thing as an `unsigned int`.

Typical usage:

```
#include <string.h>

char const * string = "This is a test";
printf("String length is %d\n", strlen(string));
```

2A.2 Copying strings: `strncpy`

It is not possible to copy strings using the assignment operator, because string variables are just pointers. So for example

```
string1 = string2;
```

will just cause the variable `string1` to point to the same characters as the variable `string2`, with no actual copying taking place.

To copy strings from one place to another, `strncpy()` can be used. Its prototype in `string.h` is

```
char *strncpy(char *dest, char const *src, size_t count);
```

The function copies the string from `src` to `dest`, including the termination character. It will not copy more than `count` characters, so it will not overflow the destination buffer — but it does *not* add the termination character if it stops early because it's reached the character count limit. This means that you should always add the termination character yourself, just to make sure it exists.

Typical usage:

```
#include <string.h>

char const * string = "This is a test";
char copy[10];

strncpy(copy, string, 10); // Will not overflow the destination
copy[9] = 0;               // Reinstate the termination character
printf("Original string: %s\n", string);
printf("Copied string: %s\n", copy);
```

⚠ **Do not use:** `strcpy()`, which is similar but does not take the `num` parameter and therefore cannot be guaranteed not to overflow the destination buffer.

2A.3 Comparing strings: `strncmp`

The `strncmp()` function compares two strings lexicographically[1]. The prototype in `string.h` is

```
int strncmp(char const *lhs, char const *rhs, size_t count);
```

The function returns:

- Zero if `lhs` and `rhs` compare equal for the first `count` characters; otherwise
- A negative value if `lhs` comes before `rhs` lexicographically, or
- A positive value if `rhs` comes before `lhs` lexicographically

The `count` parameter is good for ensuring that `strncmp()` doesn't start reading memory it shouldn't if there's a termination character missing, but can also be used to compare partial strings.

ⓘ `strcmp()` also exists, which is similar but does not take the `count` parameter. Using this is not a serious security risk in most cases, but `strncmp()` is still safer if you know the maximum length of your input strings.

2A.4 Joining strings: `strncat`

The `strncat()` function joins two strings, appending one to the end of the other. The prototype in `string.h` is

```
char *strncat(char *dest, char const *src, size_t count);
```

This function appends characters from `src` to the string in `dest`, starting by overwriting the termination character of `dest`. It will copy at most `count` characters, and will then append a termination character as well.

Typical usage:

```
#include <string.h>

char first[13] = "Hello ";
char const * second = "world!";

strncat(first, second, 5);
printf("String is now: %s\n", first);
```

[1]Lexicographical ordering is like alphabetical ordering, but using the ASCII character codes. So uppercase letters come before lowercase letters, numbers come before letters, etc.

Note that this will result in the string "Hello, world" being printed because this exclamation mark was missed off due to the count parameter being equal to 5. (This is the largest number that could be used in this example without overflowing the buffer for the first string.)

⚠ **Do not use:** `strcat()`, which is similar but does not take the `count` parameter and therefore cannot be guaranteed not to overflow the destination buffer.

2A.5 Formatting strings: `snprintf`

You've used `printf()` a lot, and it has a string-formatting cousin: `snprintf()`. This can be used to build arbitrary strings without printing them.

Its prototype in `stdio.h` is quite strange because its parameters can vary, just like `printf()`, but it is:

```
int snprintf(char *buffer, size_t bufsz, char const * format, ...);
```

The first parameter is a pointer to a buffer where the generated string will be written, and the second is the amount of available space in the buffer. The remaining parameters are exactly the same as for `printf()`. The function will write at most `bufsz` characters to the buffer, and will always include a termination character.

Typical usage:

```
#include <stdio.h>

char buffer[20];
snprintf(buffer, 20, "Three numbers: %d, %d, %d\n", 5, 6, 7);
```

Note that this will result in the correctly terminated string "Three numbers: 5, 6," being created because this completely fills the 20 character buffer.

⚠ **Do not use:** `sprintf()`, which is similar but does not take the `bufsz` parameter and is therefore cannot be guaranteed not to overflow the destination buffer.