Algorithms and Data Structures

By Grady Fitzpatrick

Tools

The subject takes a little break at this point to take a look at some parts that you will need to complete projects.

C Operators That Some People Have Not Used Before

We mostly assume students taking the subject are already fully able to use C and its operators, however, sometimes this is not the case, and a number of less used operators are often not well understood.

The % Operator

Because we end up with time at this point to go over it and it only really fits into the hash table data structure area, which is covered much later in the course, it has been added in this rather awkward position. Nonetheless, the % operator is a necessary operator and being aware of how it works early on can help explain things such as how we might convert a long item into appropriate chunks (for example, in a multi-dimensional array, or in processing certain blocks).

The % operator is also known as the 'modulo' operator, its function could be described as wrapping one number around another number, and is often described as the remainder of dividing one number by the second number. For example, 20 % 3 will give 2, which could be viewed as seeing the result of 20/3 being 6 + 2/3 and then taking the "remainder", 2. Understanding how this works is important in understanding how hash functions work. It is important in a number of other contexts (such as working out digits), however those contexts don't appear in this subject as prominently as the hash function context.

The | Operator

This operator is not expected to be used very often in this subject, however it may appear later on in your studies, and understanding it is useful. The | operator is known as the 'bitwise OR' operator, as the name suggests, it compares each bit (which is a 0 or a 1) in a number with its respective bit in another number, and if either bit is a 1, will return a 1 in that place (otherwise it will put a 0 in that place). This is primarily used in flag masks. For example, if we have three flags:
ISINEXAMPLE – 1 (0b00001)
ISINTOOLSSECTION – 2 (0b00010)
ISINTERESTING – 4 (0b00100)
We could assign a flag mask, let's call it ex, that was both an example and interesting by typing:
ex = ISINEXAMPLE | ISINTERESTING;

Which would make ex = 5 (0b00101), which might make you think “Oh hey, that’s just addition, what’s the use in that”, however the value comes in when we want to make a new flag mask, ex2, that is both an example and in the tools section:
ex2 = ISINEXAMPLE | ISINTOOLSSECTION;
which would make ex = 3 (0b00011), and then we want to see whether something satisfies both the conditions ex and ex2 by making another flag mask, ex3:
ex3 = ex | ex2;
which makes ex3 = 7 (0b00111), which is what we want, rather than the addition way, which would give us 8.

You might be thinking “oh ok, this is great, now I can make all of the flag masks I want, but what can I do with them?” To which the answer would be to use …

The & Operator

The & operator when used between two numbers is known as the ‘bitwise AND’, very similar to the other bitwise operator, the ‘bitwise AND’ operator compares each bit in one number to its respective bit in the other number and then produces a number which is the collection of all the bits that were 1 in both numbers. For example, if we have two numbers defined by:
int val1 = 126;
int val2 = 25;
The value of val1 is 126 (0b0111110) and the value of val2 is 25 (0b00011001), which means that the bitwise AND of the two values:
val1 & val2
Will give the value 24 (0b00011000). Applying this to the signal mask things we mentioned earlier, if we want to check whether our values match the masks, we simply need to do a bitwise AND against the mask that we want to check, so:
val1 & ex1
would give us 4 (0b0000 0100), we could then compare the value we get against the mask, using:
(val1 & ex1) == ex1
Which in this case is false (also known as 0). Generally, we simply check for a single bit, for example:
val1 & ISINTERESTING
Which in this case would give us 4 (0b00000100), which, as it is non-zero, can be used as true in logical conditions.

You might be thinking “but I already learned about & as an address operator, what’s it doing here again?” The answer to that question is simply that because & is only ever used on the left of identifiers when used as an address operator and between two values (with no additional operators) when it is used as a bitwise AND, it is not ambiguous. It is for the very reason that it could be confusing if you saw it without understanding what it was that it has been included here.

Other Bitwise Operators

There are some other operators that you probably won't come across in code very much but are mentioned for completeness in case you wish to investigate on your own, these are the one's complement (~), which is much like a bitwise version of the logical negation operator (!), there are also the left and right shift operators (<< and >>), which shift bits in a particular direction, bitwise XOR (^) which is much like the bitwise OR and bitwise AND, it will return the set of bits which matched with OR but not AND. The bitwise XOR is probably the most important to be aware of in your code, as C does not provide operator-based power facilities and the bitwise XOR could give you strange behaviour if you aren't aware of it.

Makefiles

Makefiles are a terribly useful feature of most UNIX systems, and there is extensive documentation about how to use them if you want to do interesting things that are out of the ordinary. However usually you just want to do the ordinary; and the extensive documentation isn't as fantastic at covering that.

Targets

Makefiles are useful because they allow a number of different compilation instructions to be specified without a great deal of effort on the part of the programmer and allow easy changes in large projects. However, often in large projects we might have multiple executables in the project, or might want to build different parts of the project with different options, for example, fast building while lots of testing is going on, a verbose output build during debugging and maximum optimisation for the release versions. What makes all these features easy to use is the use of 'targets', which direct the program using the makefile how to perform the requested function.

The format of specifying a target is that a target starts on a new line, followed by a colon (:), lines for rules of the function are indented one tab-space and sit below the target. The dependencies (explained immediately after this section) sit after the colon (if they exist).

For example:

```
test1:
        echo "Test1"
test2:
        echo "Test2"
        echo "is the second target"
```

In this example, typing "make test1" would print "Test1" on the terminal, and typing "make test2" would print the two lines "Test2" and "is the second target" on the terminal.

It is worth mentioning that some editors will convert tab-spaces to a number of spaces, if your editor does this, this will break your makefile, so be careful.

Just typing 'make'

Just typing 'make' will make the first target in the makefile. This means you can add additional targets after your typical build target such as targets that clean the object files from the directory without worrying that you'll use these targets accidentally.

Dependencies

Dependencies simply determine when a file should be recompiled (or when a target should be run). There exist a number of implicit dependencies, however explicitly listing which files would cause a change in the behaviour of the module is a good practice to use in case you make changes to parts of the program that could possibly recompile in the wrong order (thereby using an old version of the object file you want to use).

When compiling a requested target, its dependency targets will be run before running its rules.

How are dependencies specified?

The list of files (or targets) that the target relies on to be up to date are listed just after the colon after the target, for example:

```
test1: test2
        echo "Test1"
```

will run test2 before running test1.

Implicit Rules

A number of rules exist that allow targets to be specified without explicitly writing the rules to perform those targets. These are typically rules such as a .c file being compiled into a .o file with the same name, for example test1.c would automatically be compiled into test1.o if 'make test1.o' was typed and test1.c was present. This is also the case for a target with a corresponding .c file present in the same directory, for example, if 'make test1' was typed and test1.c was present, test1 would be compiled using the default compiler and settings for compiling test1.c.

Dependencies for Implicit Rules

Implicit rules save us from typing a significant amount of text, however there are some reasons why we wouldn't just trust an implicit rule to give us everything we want, dependencies. The same issues with leaving out dependencies for targets arise with implicit rules, but luckily the team who developed GNU make have us covered, we can provide the target and give it dependencies, as long as it has no specified behaviour, it will run similarly to other dependencies for explicit targets. For example, in order to specify test1 relied on test2.o being compiled before it was compiled itself, we would write:

```
test1: test2.o
```

and then test2.o would be built before test1

What if I want a different rule than the implicit one?

Sometimes, you might want your target to be performed differently than the implicit rule, in order to change the behaviour; you simply need to specify its rules. Explicit rules will override the implicit rules.

Variables

The level of variables for makefiles that you are likely to want for this subject are pretty simple, on a new line, type the name of the variable and then an equals (=) sign, and then the test that you want to be substituted whenever you use that variable. A dollar sign ($) is used with the variable name in brackets in order to get the value of the variable. For example

```
CFLAGS = -Wall –Wextra –g –O0
test1:
        gcc $(CFLAGS) –o test1 test1.c
```

would be define the target test1 the same as:

```
test1:
        gcc –Wall –Wextra –g –O0 –o test1 test1.c
```

this is particularly useful when you have many different targets that rely on the same text.

Other Makefile things

Make has many more features than those mentioned here, however they aren't really used in the makefiles you are likely to be using. If you're interested in learning about these features, you can find information in the manual on the documentation site (http://www.gnu.org/software/make/manual/). You shouldn't need to read this manual to understand the makefiles in the subject, however, if the explanation there is easier for you to understand, feel free to use it.