1.4.1 Data Types

Revision sheet

1 Primitive Data Types

A formal description of the type of data being stored or manipulated in a program. Important as they determine the operations which can be performed on the data. All programming languages support the same basic data types, they might be handled differently from language to language.

1.1 Examples

Below are a number of examples of data types and their uses.

1.1.1 String

A collection of characters including spaces and common keyboard symbols. Can also contain numbers but these will be handled as text characters not numbers.

1.1.2 Integer

A whole positive or negative number. Cannot have decimal places.

1.1.3 Real

Also known as floating point numbers. They contain decimal places and can either be positive or negative.

1.1.4 Char

This is a single character; which can be any number, letter or symbol.

1.1.5 Boolean

This can only have one of two values, true or false. Different languages use different capitalisation of the values.

1.2 More complex data types

Most languages will also have a number of other data types. These could be composite types, which are made up of a number of primitive types put together (eg. an array of integers).

1.3 Pointers

These are built in data types which some languages have that are uses to point to a value or object located in computer memory.

1.4 Null

If a data type contains nothing, then it contains a null value.

2 Number Systems

There are a number of different number systems and different methods to convert between them.

2.1 Denary (Base 10)

Used most commonly, this is the one most people learn.

1000	100	10	1
4	2	5	1

The total of the numbers above would be calculated in the following way:
$4251 = (1000 \times 4) + (100 \times 2) + (10 \times 5) + (1 \times 1)$
Denary is also known as base 10, this means each column can have one of ten possible values (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

2.2 Binary (Base 2)

This is base 2, this means each column can have one of two possible values (0, 1). The columns are also different. Moving from right to left, the columns double each time.

128	64	32	16	8	4	2	1
1	0	1	1	0	0	1	1

The largest value which can be stored in binary is 11111111_2 or 255_{10}.
Binary will be looked at in greater detail throughout this revision sheet.

2.3 Hexadecimal (Base 16)

Also known as Hex. Using this method, numbers up to 255 can be stored in two characters. This is used a lot in computing, especially in graphics and website development. There are 16 columns (getting bigger in value from right to left)
F E D C B A 9 8 7 6 5 4 3 2 1 0

2.4 Converting Between Number Systems

2.4.1 Binary To Denary

Add together all the columns in which there is a 1. Using the example shown in the binary section, the total would be 179.

2.4.2 Denary To Binary

This is the reverse of binary to denary. Work from right to left seeing if the value will fit into the column, if it won't then mark down an zero and move onto the next.

2.4.3 Denary to Hex

The easiest way to do this is to go via Binary. Convert the number into binary, then split the binary into two nibbles. The values inputted in the previous step don't need to change. With the two nibbles of (4, 2, 1, 0), convert each of them back into denary, giving two individual digits, then convert each of those into Hex.

3 Negative Binary

There are two ways to represent negative binary.

3.1 Sign And Magnitude

The MSB is used to store the sign (this means it can't be used as a number). This is the easy method as +75 can be written the same as -75. For example

	-	64	32	16	8	4	2	1
+75	0	1	0	0	1	0	1	1
-75	1	1	0	0	1	0	1	1

Sign and Magnitude has problems: the size of the number is what it was before as we only have one less bit; there are two different data types in one number (computer struggles to understand this); more difficult for computer to do calculations.

3.2 2's Complement

In this method, the most significant bit becomes -128. Then we add each of the other bits to the MSB value to reach our desired value. For example -75 in 2's complement can be seen below

	-128	64	32	16	8	4	2	1
-75	1	0	1	1	0	1	0	1

4 Binary Maths

To do maths with floating point binary, the binary point has to be in the same place for all numbers involved.

4.1 Binary Addition

This works in a very similar way to denary addition. It is shown below

	128	64	32	16	8	4	2	1
n1	0	1	1	0	0	1	1	1
+n2	0	0	0	1	0	1	1	0
result	0	1	1	1	1	1	0	1
carry					1	1		

There are three basic cases. These cases can be expanded for as many numbers as needed.
1+0=1
1+1=0 carry 1
1+1+1 = 1 carry 1

4.1.1 Overflow Errors

These are caused when there is a 1 in the carry for the furthest left hand column.

4.2 Binary Subtraction

There are basic rules to binary subtraction
0-0=0
1-0=1
1-1=0
0-1=1 (borrow 1)

5 Decimal Binary Numbers

There are two ways to represent decimal numbers in binary.

5.1 Fixed Point

8	4	2	1	.	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$
0	1	0	1	.	1	1	0	0

The example above shows an 8-bit representation of fixed point binary with the value 5.75. Exactly the same maths and logical operations can be performed on this as can be done with integer binary - it is important to make sure that the binary point is in the same place in both numbers though. To convert denary into fixed point binary is done exactly the same as with integer binary. There can be as many or as few bits before and after the binary point.

5.2 Floating Point

This is similar to fixed point, in that you can represent decimal numbers in binary however floating point has greater range and precision as the binary point can move. There are two parts to a floating point binary number - the mantissa and the exponent. The mantissa is the number itself and the exponent is the value which the binary point has to be moved by to get to the original number. The binary point, unless specified, will always be between the left most and second left most bit of the mantissa.
For example, the floating point binary number `0.101101 0011` has a 7 bit mantissa and 4 bit exponent. The exponent has a value of 3 in denary therefore we need to move the binary point three places to the right, giving us `0101.101`. We now treat the mantissa as we would any other binary number to convert to denary, giving us `5.625`. This process is the same for a negative exponent, except for the fact that the binary point moves left.

5.2.1 Normalisation

Normalisation is the process of moving the binary point so the most precise number possible can be achieved. This is achieved using the same method outlined above, but in reverse. A positive normalised mantissa will always start with `0.1` and a negative normalised mantissa will always start with `1.0`.

5.2.2 Maths

Floating point addition and subtraction work exactly the same as standard binary addition and subtraction, however the binary point needs to line up in the two numbers.

6 Bitwise Manipulation And Masks

6.1 Shifts

Binary numbers can be multiplied together or divided using a combination of shifts and addition.

6.1.1 Logical Shift

All the bits move left or right. The example below shows a logical right shift which forces the least significant bit into a 'carry bit' and the most significant bit padded out with a 0. Logical right shifts are useful for examining the contents of the least significant bit where it can be tested then the program branched.

									carry bit
before	1	0	1	1	0	0	0	1	-
after	0	1	0	1	1	0	0	0	1

6.1.2 Arithmetic Shift

This is fundamentally the same as a logical shift except the sign bit stays the same.

									carry bit
before	1	0	1	1	0	0	0	1	-
after	1	1	0	1	1	0	0	0	1

6.1.3 Circular Shifts

This is useful for performing shifts in multiple bytes. In a right circular shift, the value in the least significant bit is moved into the carry bit and the carry bit is moved into the most significant bit.

									carry bit
before	1	0	0	0	1	1	0	1	0
after	0	1	0	0	0	1	1	0	1

6.2 Masks

The output from a mask would be the same as if the boolean number was applied to that logic gate in individual bits.

		NOT	AND	OR	XOR
Input	A	1010	1010	1010	1010
Input	B		1100	1100	1100
Result		0101	1000	1110	0110

7 Character Sets

A character set is *a complete set of the characters and their number codes that can be recognised by a computer system.* There are two character sets which are commonly used: ASCII and Unicode.

7.1 The Need For Character Sets

Character sets are needed so that many different computers or connected devices are able to interpret messages and data transmitted between them, and translate them back to the original message.

7.2 ASCII

The *American Standard Code for Information Interchange* was one of the first mainstream character sets adopted into computing. It uses 7 bits, representing 128 different values. The first 32 of these values are non-printing control characters (eg, `space` or `NULL`). The remainder of the characters are used for printing characters (eg, `A` or `-`), apart from the last one which is the `DEL` character.

7.2.1 8-Bit ASCII

After some time, a larger character set was needed, the solution to this was to add a leading zero to all of the 7-bit ASCII character values. This turned ASCII into an 8-bit character set and allowed another 128 characters to be included in the set. This still wasn't big enough - Unicode was developed to solve this problem.

7.3 Unicode

By the 1980s, around the world there were several character sets which were being developed independently and were not compatible with each other. A new character set was developed by the Unicode Consortium, called Unicode (UTF-16) which is 16 bits and can represent 65536 different characters. The first 128 characters were kept the same for backwards compatibility with ASCII. A further 32-bit (UTF-32) Unicode character set was also developed, this had more characters in it. Whilst the addition of a globally recognised character set was good; the new character set had an increased size therefore files using it would have an increased sizes therefore transmission of these files would take longer.