

Data representation

A-Level Computer Science 9618 · Topic 13 revision notes

Everything from the 4 lessons of this topic on a few pages: the key idea of each lesson, the diagram to remember, the points to learn and the traps examiners set. Read it, cover it, test yourself.

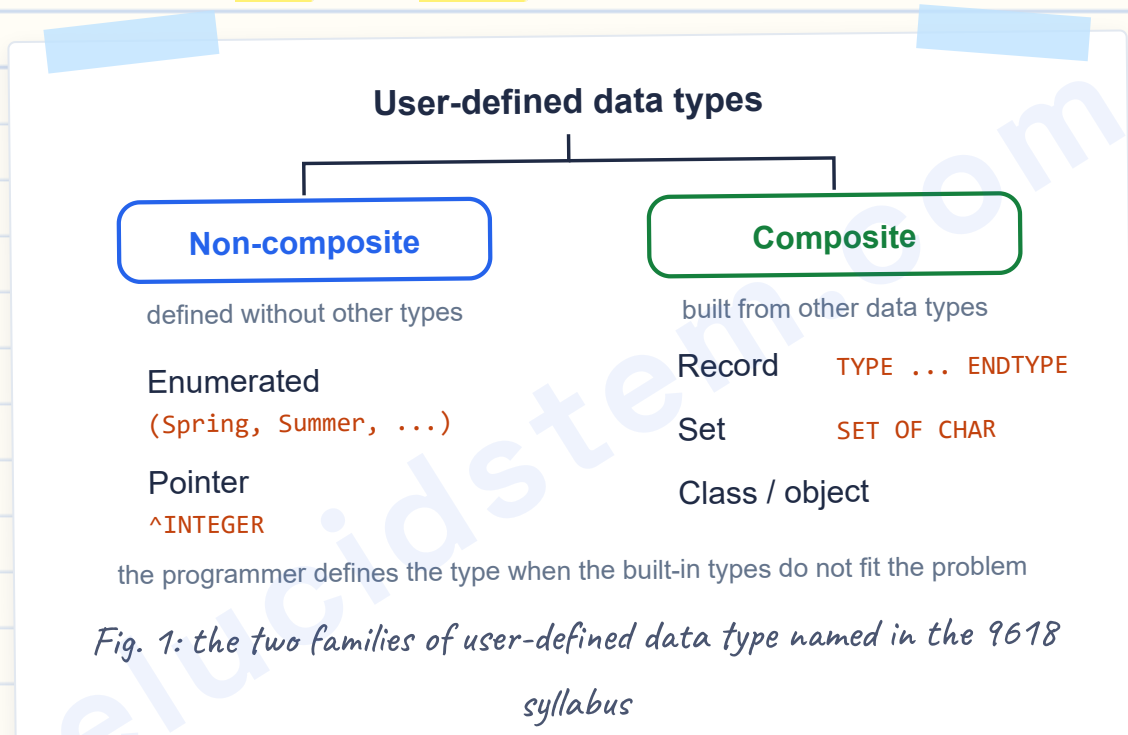
1. User-defined data types
2. File organisation, access and hashing.
3. Floating-point binary representation
4. Normalisation, range and precision

thelucidstem.com

1 User-defined data types

Syllabus 13.1

A **user-defined data type** is one the programmer defines, derived from the built-in types, because the built-in types alone do not model the data in the problem. A **non-composite** type is defined without reference to other user types: an **enumerated** type lists every value it can take, in order, and a **pointer** type holds the **memory address** of a value of a stated type. A **composite** type groups other data types under one identifier: a **record** (fields of different types), a **set** (unordered, no duplicates, supports set operations such as membership) and a **class**, whose **objects** combine attributes with methods.



Worked example: Trace a pointer

Using the code above, what are the values of `Score` and `MyPointer^` after the last line? What would OUTPUT `MyPointer` display?

- `Score ← 40` stores 40 in the memory location called `Score`.
- `MyPointer ← ^Score` stores the address of `Score` in `MyPointer`, not the value 40.
- `MyPointer^ ← MyPointer^ + 2` reads the value at that address (40), adds 2 and writes 42 back to the same location, which is `Score`.
- So `Score` and `MyPointer^` are both 42. OUTPUT `MyPointer` would display a memory address, not 42.

Answer: `Score` = 42 and `MyPointer^` = 42; `MyPointer` holds the address of `Score`

Shape, then syntax.

- ✓ **Non-composite:** enumerated `TYPE T = (A, B, C);` pointer `TYPE P = ^INTEGER;`
- ✓ **Pointers:** `^X` is the address of X; `Ptr^` is the value at the address in `Ptr`.
- ✓ **Composite:** record (`TYPE ... ENDTYPE`), set (`SET OF + DEFINE`), class (`CLASS ... ENDCLASS, NEW`).
- ✓ **Choose by shape:** fixed list of values, address, mixed fields, membership only, data plus methods.

Watch out!

Writing `TYPE Season = ("Spring", "Summer", ...)`

loses the mark. Enumerated values are not strings, so they have no quotation marks.

A set is not an array: it has no index and no order, and adding a value that is already present changes nothing. Do not describe a set as "a list of values with positions".

2 File organisation, access and hashing

Syllabus 13.2

File organisation is how records are arranged in a file. In a **serial** file records are stored in the **order they arrive**; in a **sequential** file they are stored in order of a **key field**; in a **random** file each record is stored at an **address calculated from its key** by a **hashing algorithm**. **File access** is how a record is reached: **sequential access** reads records one after another from the start (serial and sequential files), while **direct access** goes straight to the record (random files via the hash, sequential files via an index). When two keys hash to the same address there is a **collision**, handled by storing the record in the **next free address** or in an **overflow area**.

Serial

order of arrival

317	102	455	208	131
-----	-----	-----	-----	-----

both read from the start, one record after another

Sequential

in key order

102	131	208	317	455
-----	-----	-----	-----	-----

Random

address = hash(key)

					208	
0	1	2	3	4	5	6

$$208 \text{ MOD } 7 = 5$$

a hash on the key gives the address, so the record is reached directly

Fig. 1: the same five records in a serial, a sequential and a random file

Worked example: Choose the organisation

An electricity company produces a bill for every customer once a month. Its call centre must also show any single customer's account within a second. Which organisation and access method suits each task?

1. Monthly billing processes almost every record (a high **hit rate**), in key order, in one batch run.
2. So a **sequential** file accessed **sequentially** suits billing: no time is wasted calculating addresses.
3. The call centre needs one record at a time, quickly, with a low hit rate per run.
4. So **direct access** suits it: a **random** file with a hash on the account number (or an indexed sequential file).

Answer: Billing: sequential file, sequential access. Look-up: random file (or indexed sequential), direct access.

Organise, then access.

- ✓ **Serial:** order of arrival. **Sequential:** key order. **Random:** address = hash(key).
- ✓ **Sequential access** reads from the start; **direct access** jumps to the record.
- ✓ **High hit rate, batch:** sequential. **One record at a time, fast:** random.
- ✓ **Collisions:** next free address (wrap round) or an overflow area; always check the key you read.

Watch out!

Serial and sequential are not the same. A serial file has *no* order; a sequential file is sorted on a key field. Say which key the file is ordered by to earn the mark.

After hashing, always compare the key of the record found with the key wanted. The record at the home address may belong to a different key that collided there.

3 Floating-point binary representation

Syllabus 13.3

A **floating-point** number is stored as **$\text{mantissa} \times 2^{\text{exponent}}$** . The **mantissa** holds the significant digits and the **exponent** says how many places the binary point moves. In 9618 both are in **two's complement**, and the **binary point sits just after the first bit** of the mantissa, so the mantissa's place values are $-1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}$ and so on. A **positive exponent** moves the point to the right (a bigger number); a **negative exponent** moves it to the left (a smaller number). With the same number of bits, floating point gives a far wider **range** than fixed point.

8-bit mantissa and 4-bit exponent, both in two's complement



$$\text{Mantissa} = \frac{1}{2} + \frac{1}{8} + \frac{1}{16} = 0.6875$$

$$\text{Exponent} = 2 + 1 = 3$$

$$\text{Value} = 0.6875 \times 2^3 = 5.5$$

or move the point 3 places right: $0.1011 \rightarrow 101.1 = 5.5$

Fig. 1: an 8-bit mantissa and 4-bit exponent representing 5.5

Worked example: Negative mantissa

Convert the floating-point number 10100000 0010 (8-bit mantissa, 4-bit exponent, both two's complement) to denary.

1. Exponent 0010 = +2.
2. Mantissa 1.0100000: place values -1 and $\frac{1}{4}$ are set, so the mantissa is $-1 + 0.25 = -0.75$.
3. Value = $-0.75 \times 2^2 = -0.75 \times 4$.
4. Check by moving the point 2 places right: 101.00000, and in two's complement 101 is $-4 + 1 = -3$.

Answer: -3

Exponent first.

- ✓ **Value** = mantissa $\times 2^{\text{exponent}}$, both in two's complement.
- ✓ **Mantissa place values:** -1 . $\frac{1}{2}$ $\frac{1}{4}$ $\frac{1}{8}$...; the point follows the first bit.
- ✓ **To denary:** exponent, mantissa, multiply (or move the point); show each step.
- ✓ **From denary:** binary, slide the point to 0.1, count places for the exponent; negate the mantissa for negatives and check it starts 10.

Watch out!

Do not treat the mantissa as a whole number. 01011000 is not 88; the point sits after the first bit, so it is $0.1011000 = 0.6875$.

Negating a mantissa can leave it starting 11. For -4 : $4 = 0.1 \times 2^3$, so 01000000 0011 negates to 11000000 0011, which is not normalised. Shift left once and reduce the exponent; the correct answer is 10000000 0010.

4 Normalisation, range and precision

Syllabus 13.3

A floating-point number is **normalised** when the first two bits of the mantissa are **different**: a positive mantissa starts **0.1** and a negative one starts **1.0**. To normalise, **shift the mantissa left** until this is true and **subtract the number of places shifted from the exponent**, so the value is unchanged. Normalisation gives the **maximum precision** for the number of mantissa bits and a **unique representation** for each number. For a fixed total, **more mantissa bits** give **greater precision** and **more exponent bits** give a **greater range**. Many denary fractions, such as 0.1, recur in binary and can only be stored as an **approximation**, causing **rounding errors**; a result too large for the exponent causes **overflow**, and one too close to zero causes **underflow**.

normalising: same value, more significant bits kept

before 0.0011010

0101 E = 5

after 0.1101000

0011 E = 3

both are 6.5, but now the first two bits differ

positive normalised: starts 0.1

negative normalised: starts 1.0

Fig. 1: normalising 00011010 0101 gives 01101000 0011; both represent 6.5

Worked example: Positive mantissa

Normalise 00011010 0101 (8-bit mantissa, 4-bit exponent, both two's complement).

1. The mantissa starts 00, so it is not normalised.
2. Shift left until it starts 01: $00011010 \rightarrow 00110100 \rightarrow 01101000$, which is 2 places.
3. Exponent $0101 = 5$; subtract 2 to get $3 = 0011$.
4. Check: before, $0.0011010 = 0.203125 \times 2^5 = 6.5$; after, $0.1101 = 0.8125 \times 2^3 = 6.5$.

Answer: 01101000 0011

Two different bits.

- ✓ **Normalised:** positive starts 01, negative starts 10.
- ✓ **Normalise:** shift the mantissa left, subtract the shift count from the exponent; check the value is unchanged.
- ✓ **More mantissa bits:** precision up. **More exponent bits:** range up.
- ✓ **Limits:** recurring binary fractions give rounding errors; too big is overflow, too close to zero is underflow.

Watch out!

Shifting the mantissa without changing the exponent changes the number. Every left shift must take 1 off the exponent; state the new exponent in binary.

Overflow and underflow are about the size of the result, not about a negative number. Underflow means too close to zero, not "too negative".

thelucidstem.com