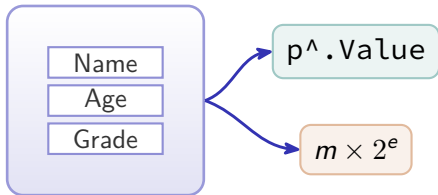


Data Representation

A-Level Computer Science ■ Topic 13

A-Level Computer Science



What we will cover

- 1 User-defined types 13.1
- 2 Pointers and linked lists 13.1.1
- 3 Files and hashing 13.2
- 4 Floating-point numbers 13.3
- 5 Range, precision, overflow 13.3.1
- 6 Recap

1

User-defined types

Sheet 13.1 – what you need to know

You must be able to:

- declare a record with an enumerated field, and assign its fields
- declare a set and say what a class adds
- tell composite from non-composite types

What the exam asks

“Write pseudocode statements to declare the composite data type, Booking...” [4]

“Identify two characteristics of the list of values given in an enumerated data type declaration.” [2]

“Complete the class diagram for Pet, to include: ...” [5]

Practise on exercise sheet 13.1, then try the real exam questions on past-paper sheet 13.1.

Why define your own types?

Built-in types (INTEGER, STRING, ...) are too loose for real data.

Restrict

限制

a STRING will hold
nonsense; an enumerated type will not

Group

组合

a taxi is a name *and*
a capacity *and* a flag

Name

自说明

DECLARE Taxi :
Vehicle reads
as what it is

User-defined types 用户定义类型 are needed because they make the compiler stricter and the code self-documenting.

Enumerated type – a fixed named list

An **enumerated type** 枚举类型 has values that are a **fixed list of named constants**.

A fleet of taxis

```
TYPE Vehicle = (M100, M230, T101,  
                T102, T120, T150)  
DECLARE MyTaxi : Vehicle  
MyTaxi ← T102
```

You cannot assign anything outside the list.
Internally the names are small integers.

Vehicle

M100

M230

T101

T102

T120

An enumerated type, drawn

TYPE Vehicle =

M100 M230 T101 T102 T120 T150

MyTaxi may only hold one of these named values

The values are **named constants** in a fixed order, so they can be compared and counted – but they are not strings and not integers.

An **enumerated type** 枚举类型 lists its own allowed values by name, so the compiler can reject anything else – clearer than remembering that 3 means Wednesday.

Composite types – several values, one name

A **composite type** 复合类型 groups several values under one name.

record

记录

fields of *different*
types in a TYPE
...ENDTYPE block

set

集合

unordered, unique
values: add, remove,
membership, union

class

类

attributes *and*
methods – an
object is an instance

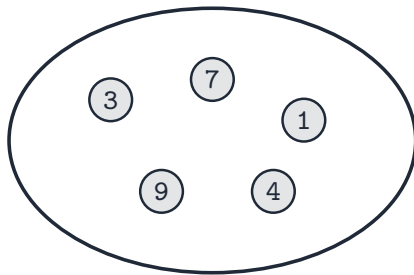
Enumerated and pointer are **non-composite**. Record, set and class are composite.

Two composite shapes

record Student

Name : STRING
Age : INTEGER
Grade : CHAR
Enrolled : BOOLEAN

one name, several fields of different types



unordered, every value unique

A **record** 记录 groups different types under one name, read by field; a **set** 集合 is unordered with no duplicates, and is only ever asked “is this in it?”

Declare a record with an enumerated field, and a set

```
TYPE Status = (Booked, Collected, Paid)
TYPE Booking
    DECLARE BookingID : STRING
    DECLARE PickUpDate : DATE
    DECLARE Passengers : INTEGER
    DECLARE Fare : REAL
    DECLARE State : Status
ENDTYPE
DECLARE Trips : ARRAY[1:100] OF Booking
Trips[1].State ← Booked
TYPE Extras = SET OF STRING
DEFINE Asked ("Seat", "Bags") : Extras
```

“Declare the composite data type, Booking...” [4]

TYPE ... ENDTYPE, one DECLARE per field, each with a *sensible type*. Dot notation reaches a field: Trips[1].Fare.

“Two characteristics of the values in an enumerated type.” [2]

They are *ordered*: Booked < Paid.

They are *identifiers*, not strings: no quotes.

A class: private attributes, public methods

Pet
Name : STRING Species : STRING Age : INTEGER
Constructor() GetName() SetAge(NewAge) Birthday()

attributes

methods

“Complete the class diagram for Pet...” [5]

Top box: the class name. Middle: **attributes** with their types. Bottom: **methods**, with a constructor and a getter or setter for each private attribute that is read or changed.

PRIVATE Age : INTEGER hides the attribute: other code must call SetAge(), which can **validate** the new value (**encapsulation** 封装).

A **class** is the template; an **object** is one instance, made by the constructor: MyPet ← NEW Pet("Rex", "dog", 3).

Choosing a type

Pick the type that matches the **kind of value**, not the first one that compiles.

a value from a fixed list

enumerated

a group of fields

record

an unordered unique collection

set

indirection / a linked structure

pointer

state *and* behaviour together

class

2

Pointers and linked lists

Sheet 13.1.1 – what you need to know

You must be able to:

- declare a pointer type and dereference it
- declare PNode and NodeType for a linked list
- traverse from HeadPointer; choose an array or a list

What the exam asks

“Identify two characteristics of the list of values given in an enumerated data type declaration.” [2]

“Write pseudocode statements to declare the composite data type, Booking...” [4]

Practise on exercise sheet 13.1.1, then try the real exam questions on past-paper sheet 13.1.

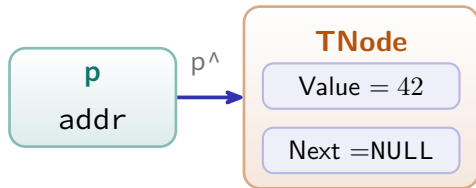
Pointer type – an address, not a value

A **pointer** 指针 holds the **memory address** of another variable (or NULL).

- builds dynamic structures: lists, trees
- passes a reference without copying
- **dereference** 解引用 $p^{\wedge}$ reaches the target

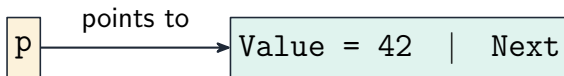
Point at a node

```
TYPE PNode = ^TNode  
DECLARE p : PNode  
p ← NEW TNode  
 $p^{\wedge}.Value \leftarrow 42$ 
```



A pointer holds an address

a pointer holds an address



$p^{\wedge}$ dereferences — reach the node's fields, e.g. $p^{\wedge}.Value$

The variable stores **where** the data is, not the data — which is what lets a linked structure grow without moving anything.

A **pointer** 指针 does not hold the data; it holds **where the data is**. Follow it to reach the value — and a pointer to nothing is null.

The linked list in pseudocode: two declarations, one loop

```
TYPE PNode = ^NodeType
TYPE NodeType
    DECLARE Data : STRING
    DECLARE Next : PNode
ENDTYPE
DECLARE HeadPointer : PNode
DECLARE Current : PNode

Current ← HeadPointer
WHILE Current <> NULL
    OUTPUT Current^.Data
    Current ← Current^.Next
ENDWHILE
```

PNode is a **pointer type**: its values are addresses of NodeType records. Current^.Data follows the pointer to the record.

Insert a new node at the front

1. Set the new node's Next to HeadPointer.
 2. Set HeadPointer to the new node.
- Swap the order and the rest of the list is lost.

An empty list has HeadPointer = NULL; the last node's Next is NULL.

Array of records or linked list?

array of records	linked list
fixed size, declared in advance	grows and shrinks as nodes are added
direct access by index: <code>Stock[7]</code>	follow the pointers from the head
insert in order: shift the items along	insert: change two pointers
no extra storage	a pointer in every node

Choose the list when the amount of data changes, or items are often inserted in order.

3

Files and hashing

Sheet 13.2 – what you need to know

You must be able to:

- describe serial, sequential and random organisation
- hash a numeric key with MOD and resolve a collision
- read and write a random file with SEEK, GETRECORD, PUTRECORD

What the exam asks

“Write program code for GetRecord()” [5]

“Explain what is meant by a hashing algorithm in the context of file access.” [3]

“Describe serial file organisation as a method of storing data records in a file.” [2]

Practise on exercise sheet 13.2, then try the real exam questions on past-paper sheet 13.2.

Three ways to lay records out

File organisation 文件组织 is the layout; **file access** is how you reach a record.

serial

串行文件

records in the *order added*. Append is fast; search is slow. Logs.

sequential

顺序文件

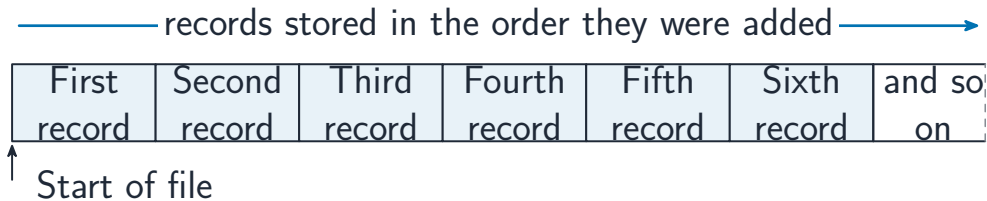
sorted on a *key*. Search can stop early; insert must shift.

random

随机文件

position from the key (often a hash). Direct lookup is fast.

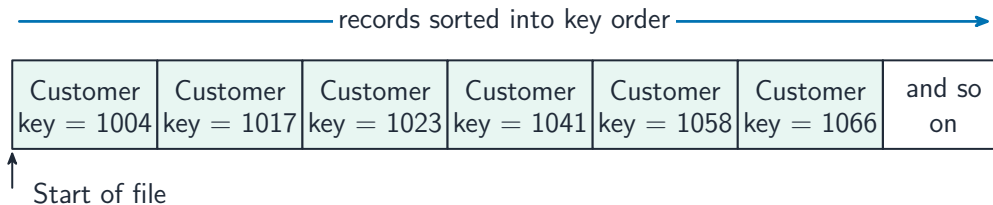
Serial – records in arrival order



New records go on the **end**. Simple to write, but any search reads from the start.

A **serial file** 顺序文件 appends each record as it arrives, so writing is fast and finding one means reading from the start.

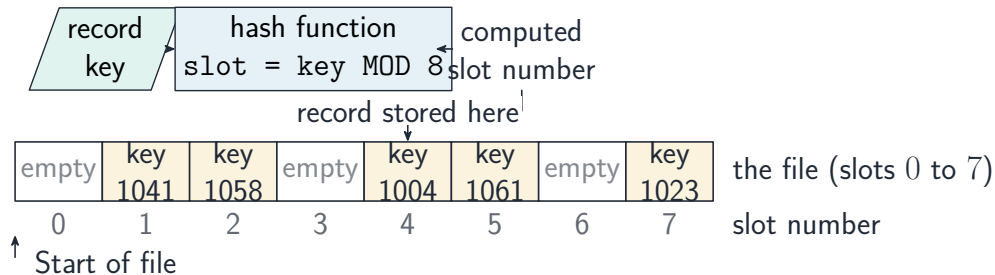
Sequential – ordered by a key



Records are kept in **key order**, so a search can stop early – but an insertion means rewriting the file.

A **sequential file** 有序文件 keeps records **sorted by a key**, so a search can stop early – but every insertion has to make room.

Random – the key computes the address



A **hash** 散列 of the key gives the record's position, so one read finds it – **direct access**, at the cost of managing collisions.

Sequential access vs direct access

- **sequential access** 顺序存取 – read from the start, one record after another
- **direct access** 直接存取 – jump straight to a known position

Serial and sequential files **need** sequential access. A random file **allows** direct access.

Match the layout to the dominant operation: single-key lookups favour random; in-order reports favour sequential.

serial / sequential

walk from the start

random

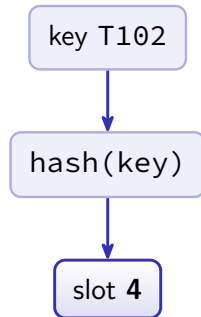
hash, then jump

A hash function turns a key into an address

A **hash function** 散列函数 takes a record key and produces the **slot** where the record is stored. A good one is fast,

deterministic 确定性, and spreads keys evenly. For N slots:

- modulo: $\text{key} \bmod N$
- folding: split, add the pieces, $\bmod N$
- string: sum character codes, $\bmod N$



Keep the **load factor** 装填因子 below about 70%.

Collisions – two keys, one slot

A **collision** 冲突 is when two keys hash to the same address.

linear probing

线性探测

use the next free slot, wrapping around. Simple, but keys *cluster*.

chaining

链接法

each slot points to a **linked list** 链表.
No clustering; uses more memory.

rehashing

再散列

apply a second hash function. Spreads keys, more work.

Worked example – resolve a collision

Keys A and B both hash to slot 2

Linear probing: B goes in the next free slot (3).

Chaining: slot 2 holds a list $A \rightarrow B$.

To search: hash the key, read that slot; if the keys do not match, follow the resolution strategy until a match or an empty slot.

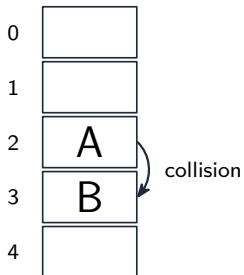


probing: B sits in 3

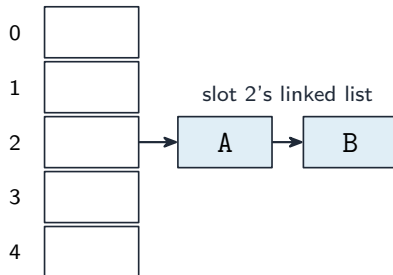
Empty slot = “not found”
when probing.

Resolving a hash collision

linear probing



chaining



Resolving a hash collision: linear probing uses the next free slot; chaining keeps a linked list per slot

Read a record from a random file

```
FUNCTION GetRecord(Key : INTEGER)
                                RETURNS Item
    DECLARE Stock : Item
    DECLARE Address : INTEGER
    Address ← Key MOD 500
    OPENFILE "Stock.dat" FOR RANDOM
    REPEAT
        Address ← Address MOD 500 + 1
        SEEK "Stock.dat", Address
        GETRECORD "Stock.dat", Stock
    UNTIL Stock.ItemID = Key
    CLOSEFILE "Stock.dat"
    RETURN Stock
ENDFUNCTION
```

Keys 1207 and 3707: both give $207 + 1 = 208$. The second is stored in the next free address, so the loop reads 208, then probes 209, 210, ... until the key matches.

To write: hash, SEEK, then PUTRECORD "Stock.dat", Stock.

MOD 500 + 1 wraps from address 500 back to 1.

Two definitions the mark scheme wants

“Explain what is meant by a hashing algorithm in the context of file access.” [3]

A calculation on the record's **key field**...
... gives the **address** where the record is stored in the file.

The same calculation finds it again: **direct access**, no search. Two keys may give one address (a **collision**).

“Describe serial file organisation...” [2]

Records are stored one after another in the order they **arrive**, in no key order.
A new record is **appended** at the end.

Serial suits a log or a transaction file, read from start to end.

4

Floating-point numbers

Sheet 13.3 – what you need to know

You must be able to:

- convert a normalised float to denary
- store a positive or negative denary value, normalised
- work in formats other than 8+8 bits

What the exam asks

“Calculate the normalised binary floating-point representation of -76.1875 in this system. Show your working.” [4]

“Calculate the denary value of the given normalised binary floating-point number. Show your working.” [3]

“Write the normalised floating-point representation of the following positive binary number...” [2]

Mantissa $\times$ two-to-the-exponent

A **floating-point** 浮点 number is binary scientific notation:

$$\text{value} = \text{mantissa} \times 2^{\text{exponent}}$$

Both fields are **two's complement** 补码 integers.

- more mantissa bits $\Rightarrow$ more **precision**
- more exponent bits $\Rightarrow$ more **range**

The mantissa is a binary *fraction*: the first bit after the point is $\frac{1}{2}$, then $\frac{1}{4}$, $\frac{1}{8}$, ...

mantissa

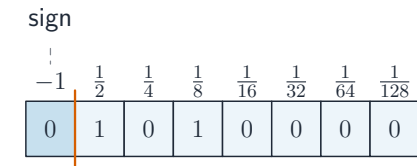
0.1010000

exponent

00000010

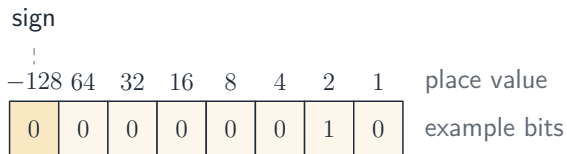
$$0.625 \times 2^2 = 2.5$$

Floating-point numbers



implied
binary point

mantissa (8 bits)



exponent (8 bits)

The place values of an 8-bit mantissa and an 8-bit exponent

Worked example – binary to denary

Mantissa 10110000, exponent 00000011

Exponent 00000011 = +3.

Mantissa starts with 1, so it is **negative**. Sign bit = -1, fraction $\frac{1}{4} + \frac{1}{8} = 0.375$:

$$m = -1 + 0.375 = -0.625$$

$$\text{value} = -0.625 \times 2^3 = -5.0$$

Converting binary → denary

- 1 exponent as a signed integer
- 2 mantissa as a fraction (two's complement if it starts with 1)
- 3 multiply by 2^{exponent}

Worked example – store +2.5

Denary → **binary**: write the number as a binary fraction $\times$ a power of 2, then store mantissa and exponent.

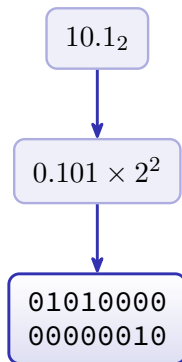
Write +2.5 in this 8+8-bit format

In binary $2.5 = 10.1$.

As a normalised fraction: $2.5 = 0.101 \times 2^2$.

mantissa **01010000** (sign 0, then .101)

exponent **00000010** ($= 2$)



Worked example – store -76.1875 in 12+4 bits

12-bit mantissa, 4-bit exponent

1. $+76.1875 = 1001100.0011$ ($64 + 8 + 4, \frac{1}{8} + \frac{1}{16}$)
2. Move the point 7 places: 0.10011000011×2^7
3. Positive mantissa 010011000011
4. Negate (flip, add 1): 101100111101
5. Exponent $7 = 0111$

Check: $-1 + 0.40478515625 = -0.59521484375; \times 2^7 = -76.1875$.

A normalised **negative** mantissa starts 10; a positive one starts 01.

Normalisation maximises precision

A number is **normalised** 规格化 when the first significant bit sits **immediately after the binary point** – no wasted leading zeros.

- positive: starts 0.1 ...
- negative (two's complement): starts 1.0 ...

Shift the mantissa left and decrease the exponent (or right and increase it) until that pattern holds. The value is unchanged.

Normalise 0.0101

Shift left until it reads 0.1 ...:

$$0.0101 \rightarrow 0.101 \times 2^{-1}$$

mantissa 0.101, exponent -1

Each **left** shift subtracts one from the exponent.

What normalising does to the bits

before 0.0011010 exponent 4 (two wasted leading zeros)

 ↓ shift left 2, exponent - 2

after 0.1101000 exponent 2 (normalised — same value)

Shifting until the first two bits differ uses every mantissa bit for **precision** instead of for leading zeros.

Normalising shifts the mantissa until the first two bits **differ**, adjusting the exponent to match. That uses every bit of precision the mantissa has.

5

Range, precision, overflow

Sheet 13.3.1 – what you need to know

You must be able to:

- split 16 bits between mantissa and exponent, and justify it
- explain overflow and underflow from the exponent
- explain rounding errors, and why money is not stored as a float

What the exam asks

“Calculate the normalised binary floating-point representation of $-76.1875\dots$ ” [4]

“Calculate the denary value of the given normalised binary floating-point number.” [3]

Practise on exercise sheet 13.3.1, then try the real exam questions on past-paper sheet 13.3.

The same 16 bits, three splits: range or precision

split	max exp.	largest	fraction bits
12 + 4	7	≈ 128	11
10 + 6	31	$\approx 2.1 \times 10^9$	9
8 + 8	127	$\approx 1.7 \times 10^{38}$	7

More **mantissa** bits: more **precision**. More **exponent** bits: more **range**.

Both come from the exponent

Overflow: the result needs an exponent larger than the largest one (7 in 12+4).

Underflow: the result is so close to 0 that it needs an exponent below the smallest (-8); it is stored as 0.

Money is stored as whole cents (INTEGER): 0.10 has no exact binary fraction.

Approximation and rounding errors

Many denary reals **cannot be stored exactly** in binary:

$$0.1_{10} = 0.\overline{00011}0011\dots_2$$

it recurs for ever, so the mantissa cuts it off

- **rounding errors** 舍入误差 build up: $0.1 + 0.2$ is not exactly 0.3
- **comparisons fail**: test $\text{ABS}(x - 0.3) < 1\text{e-}9$, not $x = 0.3$
- **overflow** 溢出 / **underflow** 下溢: the exponent runs out of range

Why $0.1 + 0.2$ is not 0.3

Many denary reals **cannot be stored exactly** in binary – $0.1_{10} = 0.000110011\dots_2$ must be truncated.

rounding

舍入误差

errors build up over
many operations

overflow

溢出

result too large for
the exponent's range

underflow

下溢

result too small,
rounds to zero

Never test reals with $=$. Use a tolerance, or use **fixed-point** 定点 / BCD for money.

6

Recap

Exam tips

- For **floating-point** binary $\rightarrow$ denary, read the mantissa as a fraction and the exponent as a signed integer, then multiply; a **negative mantissa** follows two's-complement rules.
- **Normalise** by shifting until the first digit after the point differs from the sign bit – this maximises precision.
- Explain **rounding, overflow and underflow** errors and why 0.1 cannot be stored exactly.
- Compare file organisation (serial, sequential, direct) and how **hashing** finds a record quickly.

Topic 13 – key takeaways

1 Types

enumerated, pointer, record, set, class

2 Files

serial, sequential, random; sequential vs direct access

3 Hashing

key $\rightarrow$ address; probing / chaining / rehash

4 Float

$m \times 2^e$; normalise; rounding / overflow

Check yourself

- 1 Which file type gives direct access by key?
- 2 Linear probing vs chaining – what is the trade-off?
- 3 How does a normalised *positive* mantissa start?
- 4 Why can 0.1_{10} not be stored exactly?

Answers 1. random 2. probing clusters; chaining uses extra memory 3. $0.1 \dots$ 4. it is a repeating binary fraction